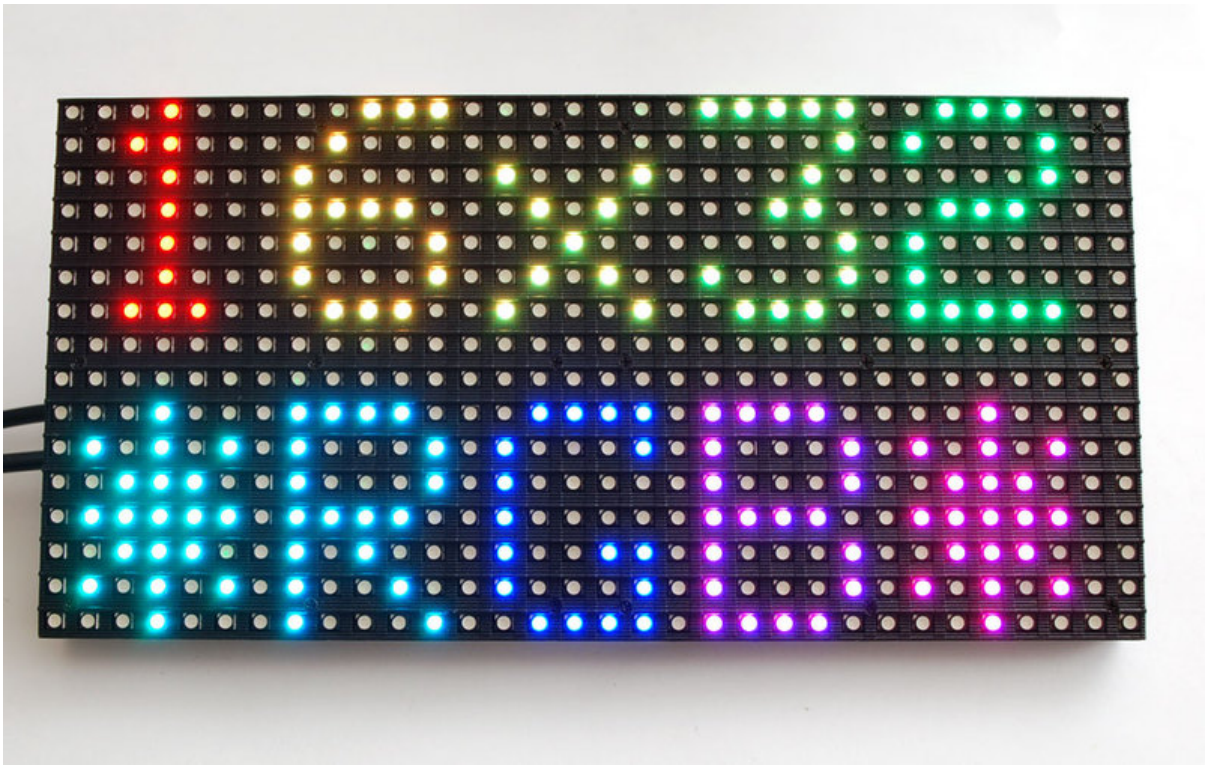
RGB LED Matrix Basics

Created by Phillip Burgess



https://learn.adafruit.com/32x16-32x32-rgb-led-matrix

Last updated on 2024-06-03 01:13:42 PM EDT

Table of Contents

Overview 3

- COMPATIBLE HARDWARE
- COMPATIBLE SOFTWARE

Power 6

Connections 9

- Connecting to Arduino

Connecting Using RGB Matrix Shield 12

- Assembling the Partially Assembled Version
- Assembling the Original Version
- Powering the Shield
- Metro M4 Usage

Connecting with Jumper Wires 15

- Connect Ground Wires
- Upper RGB Data
- Lower RGB Data
- Row Select Lines
- LAT Wire
- OE Wire
- CLK Wire

Connecting Using a Proto Shield 26

- Connect Ground Wires
- Upper RGB Data
- Lower RGB Data
- Row Select Lines
- LAT Wire
- OE Wire
- CLK Wire

Test Example Code 32

RGBMatrix Library 35

How the Matrix Works 37

FAQ 39

- Wiring Issue Examples
- Ghosting Issue Examples

Downloads 42

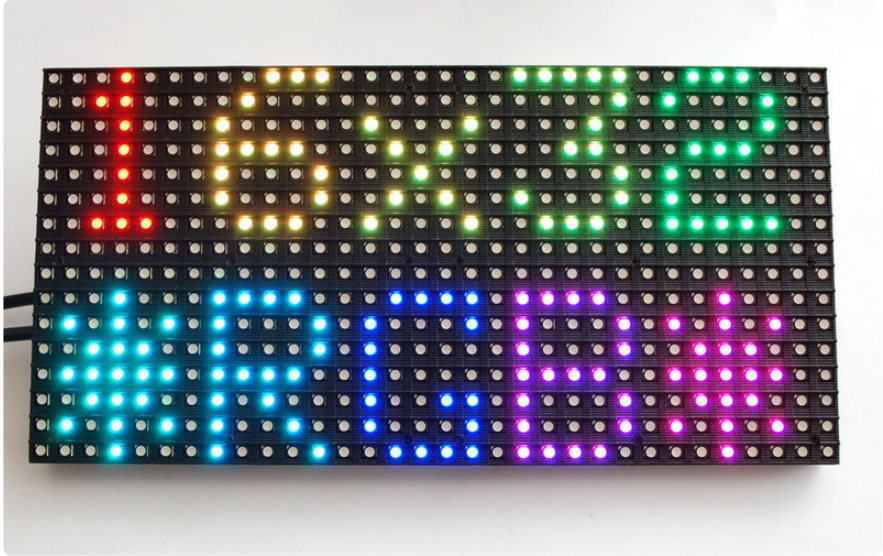
- Files
- Schematic and Fab Print for Partially Assembled Version
- Schematic & Fabrication Print Original Version

Overview

This guide is for boards in the **ARDUINO ecosystem**. We have a different guide for Raspberry Pi (https://adafru.it/kdh), and another for CircuitPython (https://adafru.it/L7b).

Arduino Uno is limited to 32x16 pixels, single-buffered.

We recommend the Adafruit Protomatter library for more modern microcontroller boards. See COMPATIBLE SOFTWARE below.



Bring a little bit of Times Square into your home with our 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.

COMPATIBLE HARDWARE

The following boards are **plug-and-play ready** with the RGB Matrix Shield and software mentioned in this guide:

- **Adafruit Metro M0**
- **Arduino Zero**

- **Arduino Uno** (or compatible ATmega328P boards) — **limited to 32x16 matrix**, no double-buffering (needed for flicker-free animation)

The following are supported by the software, but require **additional wiring or jumpers** to use the RGB Matrix Shield:

- **Arduino Mega** (or compatible ATmega2560 boards)
- **Adafruit Metro M4**

The following are **NOT supported** by the software or shield:

- **Arduino Leonardo** (or compatible ATmega32U4 boards) (though the same form factor, the Uno and Leonardo route different shield connections to the microcontroller)
- **Netduino** and other Arduino-alikes not mentioned in above lists (but other libraries or shields might exist elsewhere)
- **Teensy** (but see the **SmartLED** Shields and software for Teensy 3.X and 4.X, which provide excellent performance)
- **Raspberry Pi** (but there are different Bonnets (https://adafru.it/AJp) and HATs (https://adafru.it/kdh) for this)

COMPATIBLE SOFTWARE

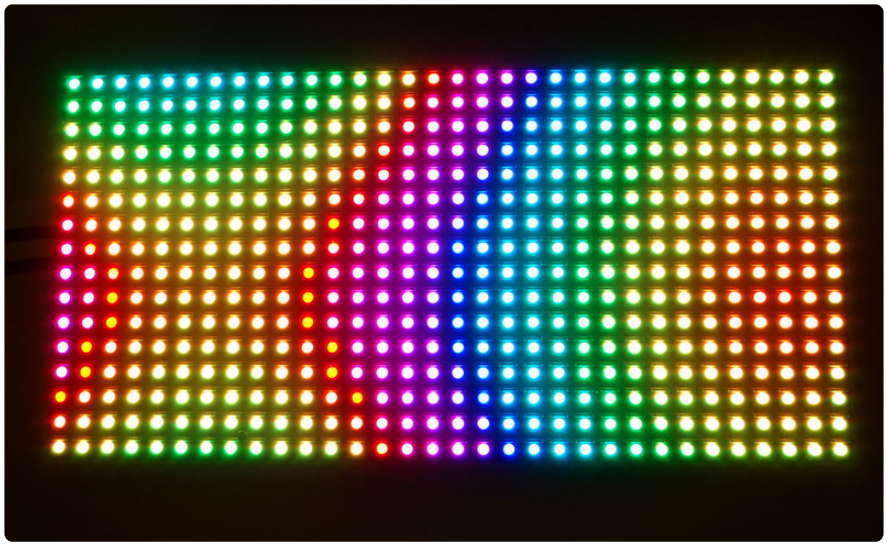
The software support for driving the RGB matrix panels has evolved with time and the availability of newer, more powerful Arduino boards. **There are currently two different Arduino libraries.** Which library to use depends on the Arduino board being used.

- RGB matrix Panel (https://adafru.it/aHj) - This is the older, original library and is covered later in this guide. Use this library with an **Arduino UNO** or **Arduino Mega**.
- Adafruit Protomatter (https://adafru.it/LaI) - This is a newer library covered in a separate guide (https://adafru.it/19fW). Use this library with an **Arduino Zero**, **Metro M0**, **Metro M4**, **RP2040** based board, **ESP32** based board, or other supported **32bit boards**. See the Protomatter guide for more details. (https://adafru.it/19fW)



These panels require 12 or 13 digital pins (6 bit data, 6 or 7 bit control) and a good 5V power supply, at least a couple amps per panel. We suggest our 2A (or larger) regulated 5V adapters 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 normally 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, we managed to squeeze 12-bit color (4096 colors) 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 most Arduino-compatible boards, you'll need 12 digital pins, and about 800 bytes of RAM to hold the 12-bit color image (double that for the 32x32 matrix, double again for smooth double-buffered animation).

The library works with a LIMITED NUMBER of boards. Please see the COMPATIBLE HARDWARE lists above.

Power

Although LEDs are very efficient light sources, get enough of them in one place and the current really adds up.

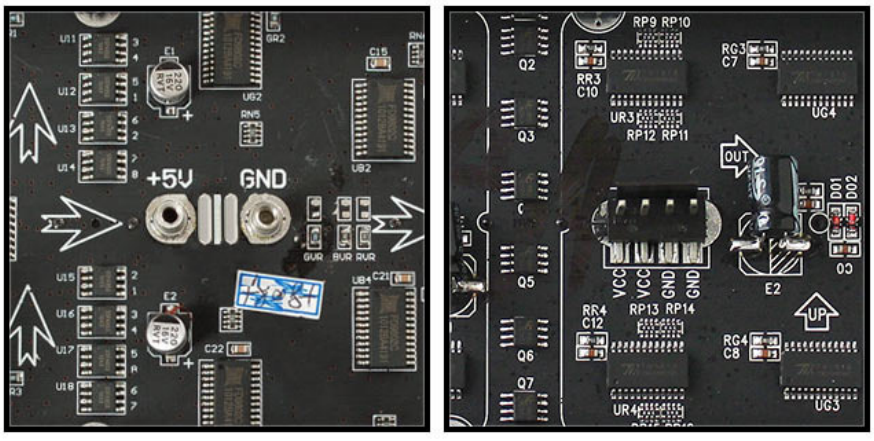
A single **32x16** or **32x32** RGB matrix, running full tilt (all pixels set white), can require nearly **4 Amps** of current! Double that figure for a **64x32** matrix.

On average though, displaying typical graphics and animation, these panels will use less...a **2A** supply is usually sufficient for a single 32x16 or 32x32 panel, or 4A for a 64x32 panel. There's no harm in using a larger power supply rated for more Amps (e.g. a 10A supply), but never use one with a higher Voltage (use **5V, period**)!

On these panels, the power connection is separate from the data connection. Let's begin by connecting 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:

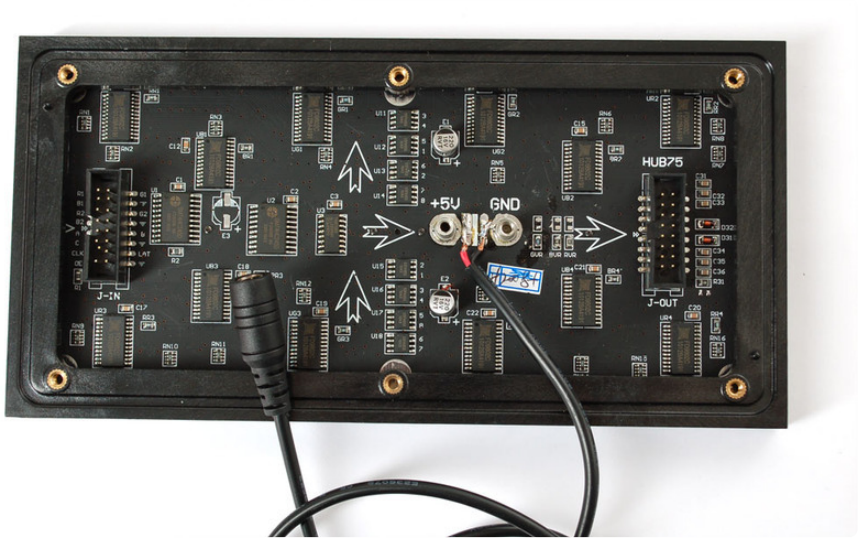


On the left is a screw post power connector (with adjacent pads for soldering wires directly). On the right, a Molex-style header. Some panels will have two headers...the power cable included with these panels has connectors for both headers.

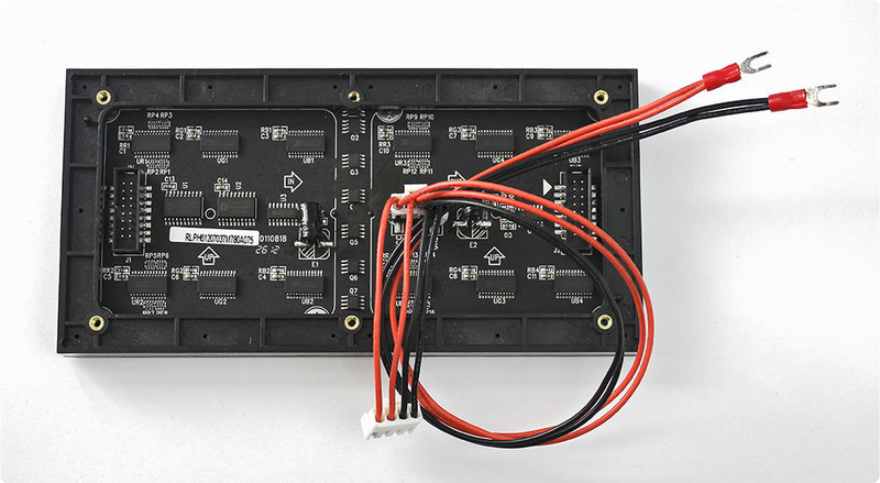
With the posts-and-pads connector, you can either screw down the spades from the power cable, or another approach is to cut a 2.1mm jack from this extension cord (http://adafru.it/327) and solder it to the pads on the panel back. This way you can plug the 5V from a wall adapter (http://adafru.it/276) 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.



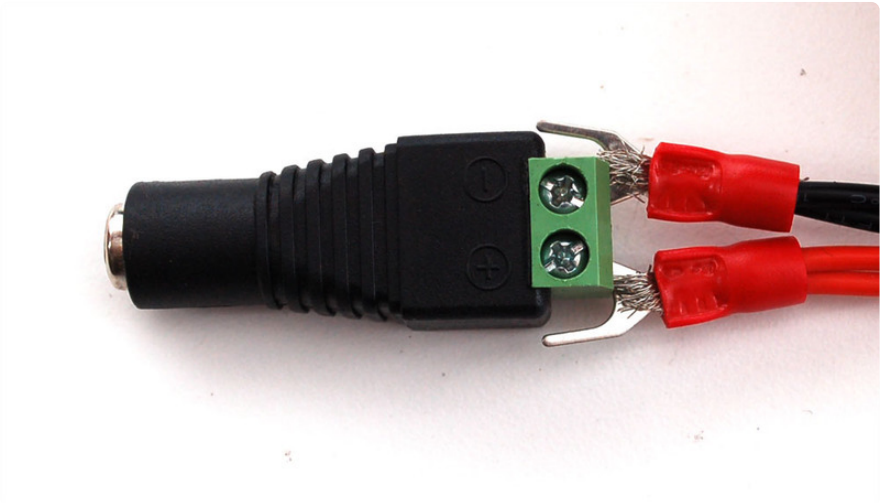
Solder both pins correctly to the power port. Make sure you get this right because there is no protection diode!



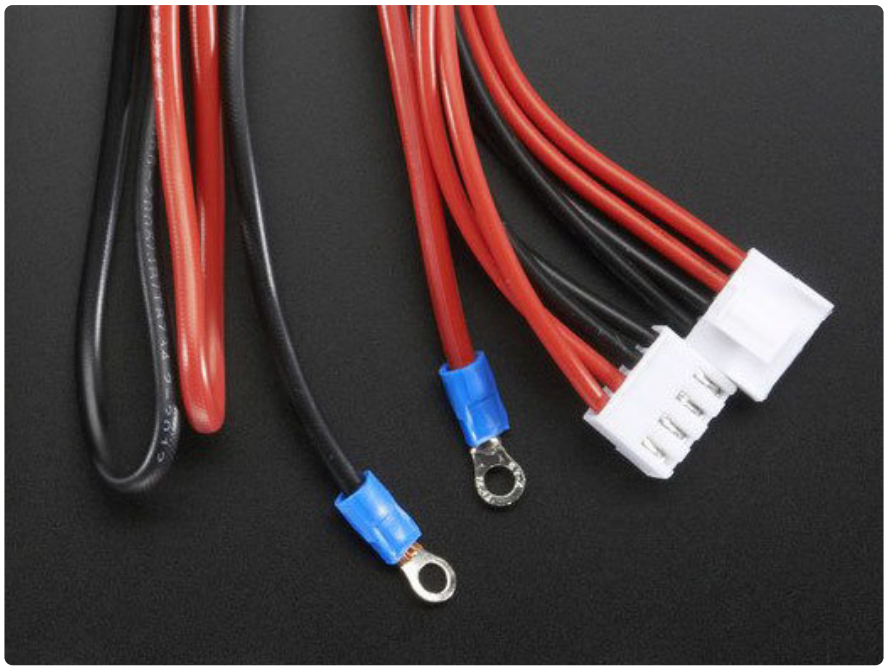
If your panel has the Molex-style header, just plug in the included power cable, observing the correct polarity.



If your power cable came with spades at the opposite end of this power cable, they can be screwed into a 2.1mm terminal block adapter. Works nicely! Don't allow the exposed connectors to contact metal though...you should probably cover this with heat-shrink tube or electrical tape.



You may receive power cables with different endings, e.g. round instead of spade ends, or maybe with another Molex connector. Just strip the cables and wire directly to the power plug



Connections

These panels are normally designed for chaining (linking end-to-end into larger displays)...the output of one panel connects to the input of the next, down the line.

With the limited RAM in an Arduino, chaining is seldom practical. Still, **it's necessary to distinguish the input and output connections** on the panel...it won't respond if we're connected to the wrong socket.

Although the panels support chaining, this is VERY impractical on Arduino-class boards and our library DOES NOT SUPPORT it. A more powerful system like a Raspberry Pi may be a better choice for chained panels!

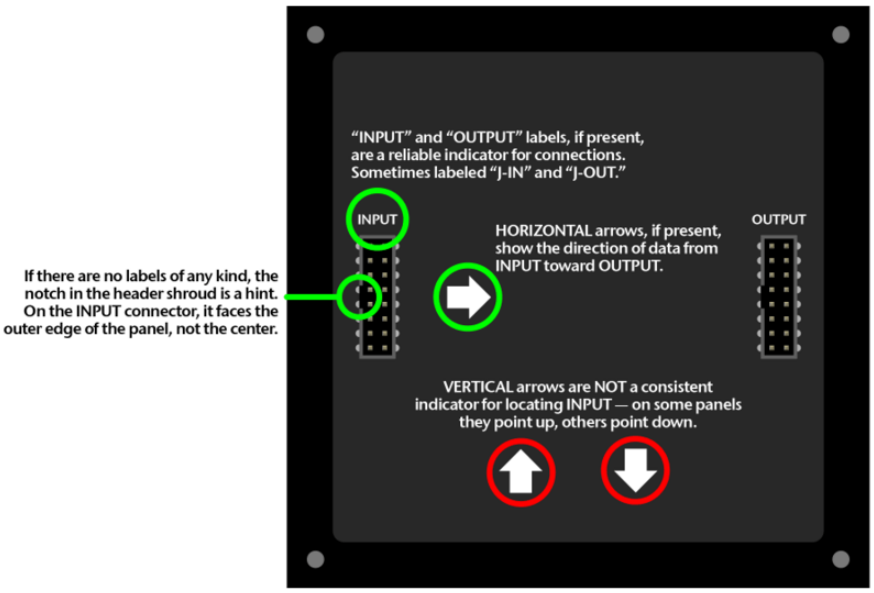
Flip the matrix over so you're looking at the back, holding it with the two sockets situated at the **left and right edges** (not top and bottom).

On some panels, if you're lucky, the sockets are labeled INPUT and OUTPUT (sometimes IN and OUT or similar), so it's obvious which is the input socket.

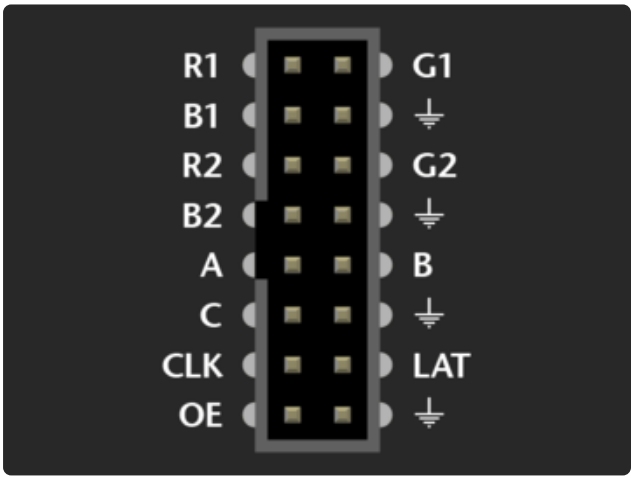
If INPUT is not labeled, look for one or more arrows pointing in the **horizontal** direction (ignore any vertical arrows, whether up or down). The horizontal arrows show the direction data moves from INPUT to OUTPUT — then you know which connector is which.

If no such labels are present, a last option is to examine the plastic shroud around the

connector pins. The key (notch) on the INPUT connector will face the outer edge of the panel (not the center).



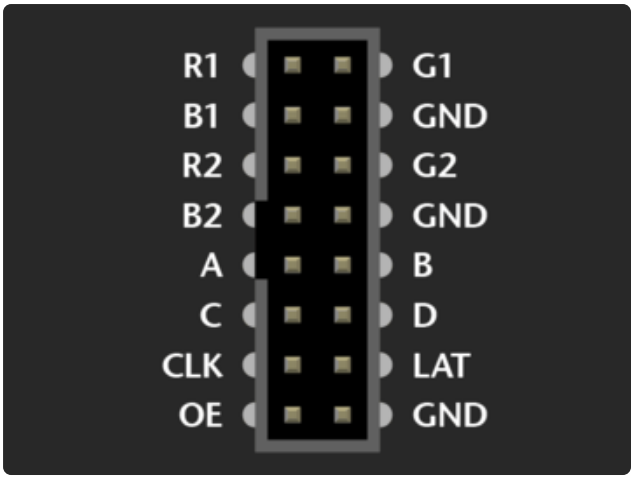
The arrangement of pins on the INPUT connector varies with matrix size and the batch in which it was produced...



A **32x16** panel uses this pin arrangement. The labels might be slightly different, or the pins might not be labeled at all...but in either case, use this image for reference.

Notice there are four ground connections. To ensure reliable performance, **all four should be connected to GND** on the Arduino! A solderless breadboard is handy for making this split.

Here's the layout for **32x32** and **64x32** panels. We'll call this "**Variant A**." Some panels use different labels, but the functions are identical.

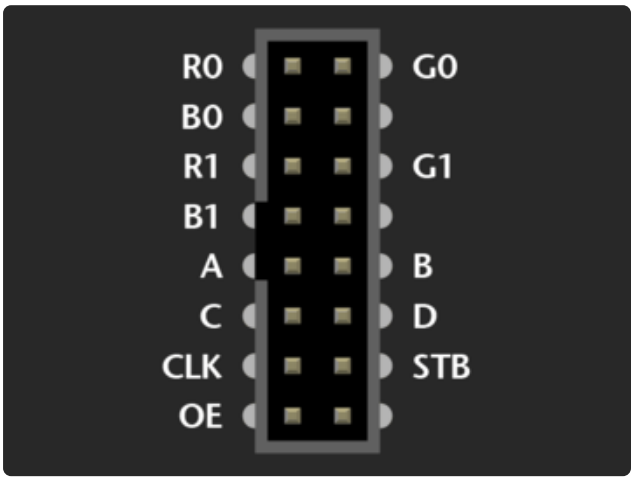


The layout is very similar to the 32x16 panel, with pin "D" replacing one ground connection.

This is the layout we'll be referencing most often.

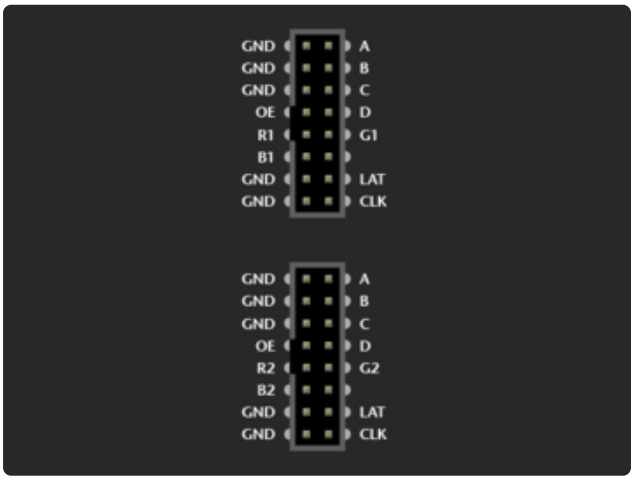
If you have a 32x32 panel with no pin labels at all, then use this layout.

"**Variant B**" for **32x32** and **64x32** panels. **The wiring is identical to Variant A above, only the labels are different.**



Ground pins aren't labeled, but still need to be connected.

LAT (latch) is labeled STB (strobe) here. R1/G1/B1/R2/G2/B2 are changed to R0/G0/B0/R1/G1/B1...but again, no functional difference, it's just ink.



Our earliest **32x32** panels had a **two-socket** design, let's call it "**Variant C**." All the same pin functions are present but the layout is very different.

R/G/B on the **upper** socket correspond to R1/G1/B1 in Variant A. R/G/B on the **lower** socket correspond to R2/G2/B2.

All the other signals (A/B/C/D/CLK/LAT/OE) need to be connected to **both** sockets — e.g. one pin on the Arduino drives both CLK pins, and so forth.

Connecting to Arduino

There are two or three methods for connecting a matrix to an Arduino:

1. **Jumper wires** inserted between Arduino headers and a ribbon cable — this works well for testing and prototyping, but is not durable.
2. The **Adafruit RGB Matrix Shield** makes connecting these panels to an Arduino as easy as can be, and is best for permanent installations.
3. One could build a **proto shield** to replicate the pinout of option #2. But given the Matrix Shield's low cost, this might not be worth the effort nowadays.

These panels are normally run by very fast processors or FPGAs, not a 16 MHz Arduino. To achieve reasonable performance in this limited environment, our software is optimized by **tying specific signals to specific Arduino pins**. A few control lines can be reconfigured, but others are very specific...**you can't wire the whole thing willy-nilly.** The next pages demonstrate compatible wiring...one using the RGB Matrix Shield, the using jumper wires.

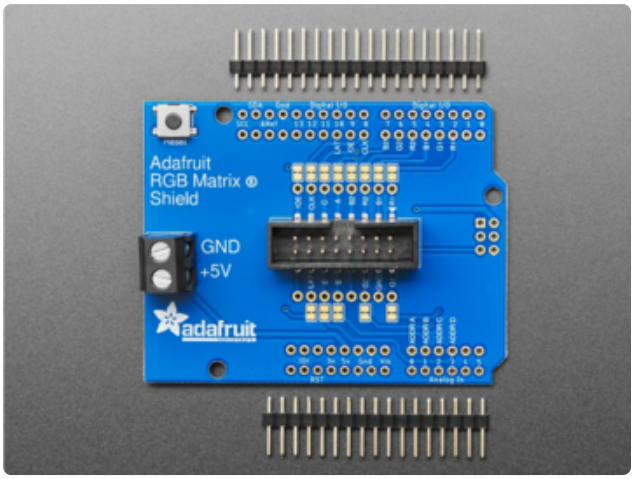
Connecting Using RGB Matrix Shield

This is the **preferred** method for pairing these matrices with an Arduino-sized board, as it's quick and trouble-free.

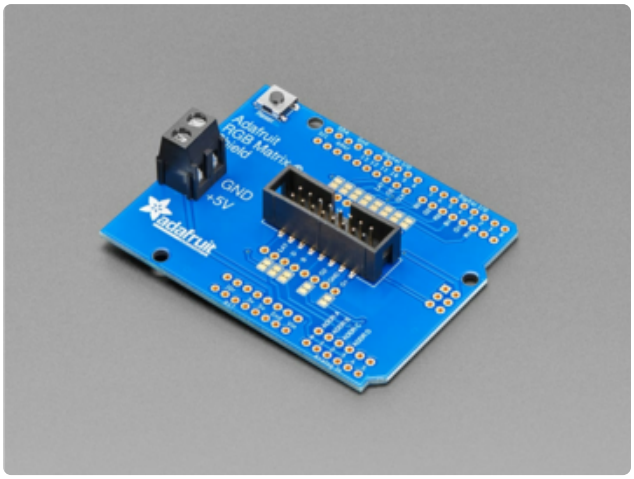
The **Adafruit RGB Matrix Shield** works with the **Arduino Uno** and **Zero**, and the **Adafruit Metro M0**...and with one or more jumper wires can work with the **Metro M4** (and potentially other boards with this form factor, if a compatible Arduino library is available).

The shield does **not** directly work with the Arduino Mega — additional jumper wires are needed to pins off the shield — see the "Jumper Wires" page for pinouts, or consider making your own Mega proto shield for that board.

Assembling the Partially Assembled Version

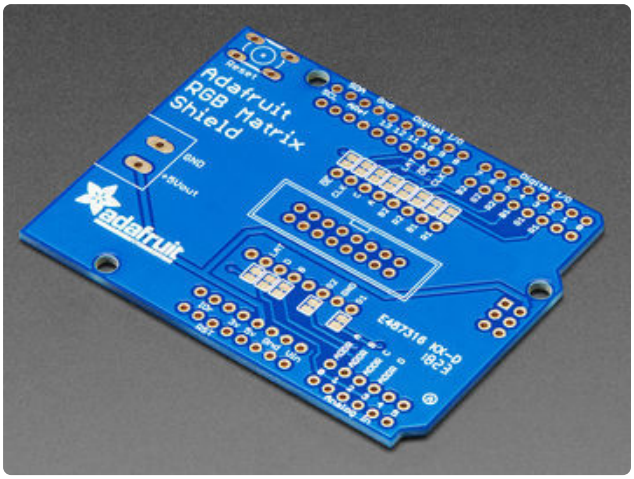


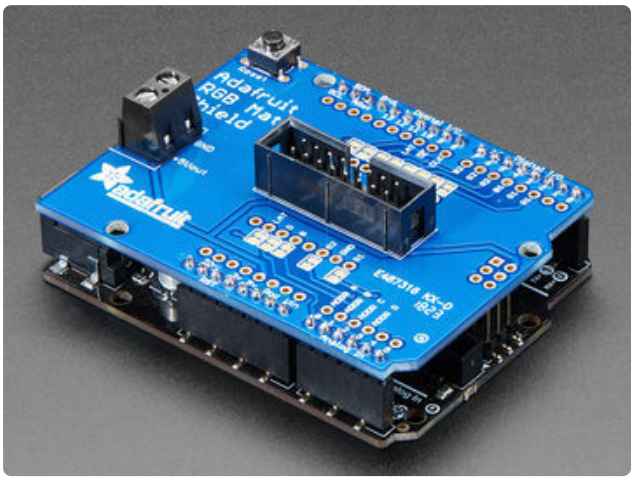
The shield comes with the button, terminal block, and the 16-pin header already soldered on. All you need to do is solder the headers!



The included headers should be installed from the underside and soldered from the top.

Assembling the Original Version





The shield arrives unpopulated and you'll need to do a little soldering to get it going.

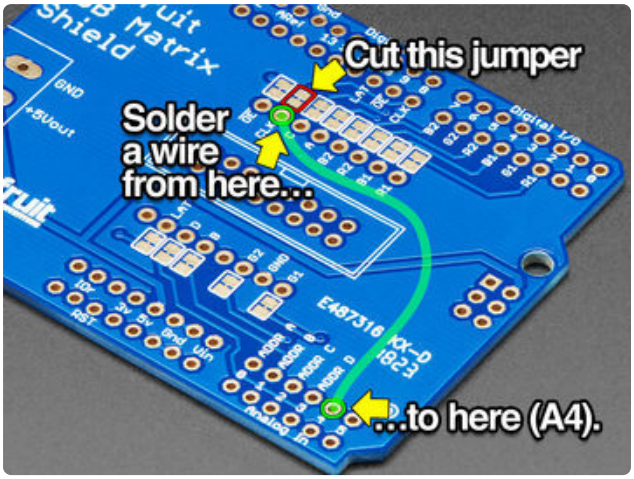
Header pins are installed from the underside and soldered on top. Three components — a button, power terminal and 16-pin header — insert from the top and are soldered underneath.

The 16-pin (8x2) header must be installed in the correct orientation! The polarity notch is indicated on the silkscreen, or you can see in the photos above that the notch faces the digital I/O pins. If you install this backwards the matrix will not work!

Powering the Shield

Power to the LED matrix can be connected to the shield's screw terminals — red wire to +5Vout, black wire to GND — and the whole circuit is then powered from the Arduino's DC jack or a USB cable at a safe and regulated 5 Volts.

Metro M4 Usage



The shield requires a small modification to work with the Adafruit **Metro M4**:

Use a small file or hobby knife to cut the PCB trace between the two pads indicated here.
Solder a wire from the adjacent “CLK” pin to the “Analog In 4” pin.

A corresponding change is required in one’s code — look for the “CLK” pin definition in any of the matrix examples...

```
#define CLK 8
```

And change the “8” to “A4”:

```
#define CLK A4
```

If you have a Metro M4 Airlift - make sure you select the right board in the Tools dropdown. Many folks select Metro M4 by accident (or vice versa)

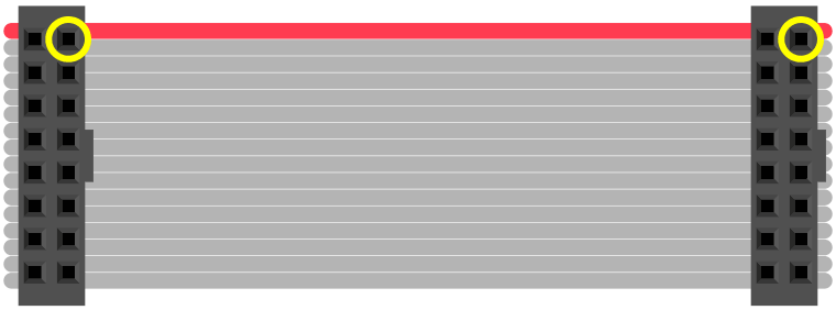
Connecting with Jumper Wires

WE DO NOT GUARANTEE SUCCESS IF USING JUMPER WIRES TO THESE PANELS. PLEASE USE A FEATHER/SHIELD/HAT SO WIRING IS DONE FOR YOU! We offer NO support when hand-wiring, ITS TOO HARD TO DEBUG!

Ribbon cables and their corresponding headers are sometimes a topological puzzle. Here’s a trick to help keep track...

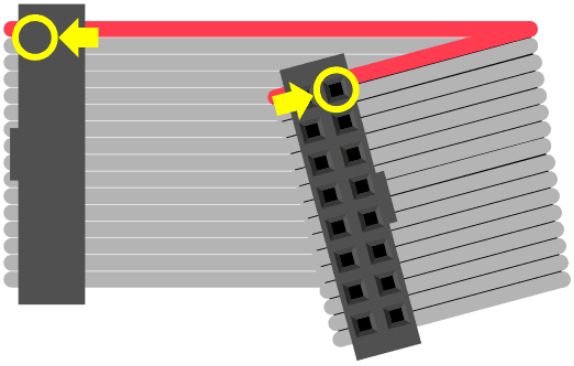
If you hold the ribbon cable flat — no folds — and with both connectors facing you, keys pointed the same direction — now there is a 1:1 correlation between the pins. The top-right pin on one plug links to the top-right on the other plug, and so forth. This holds true even if the cable has a doubled-over strain relief. **As long as the**

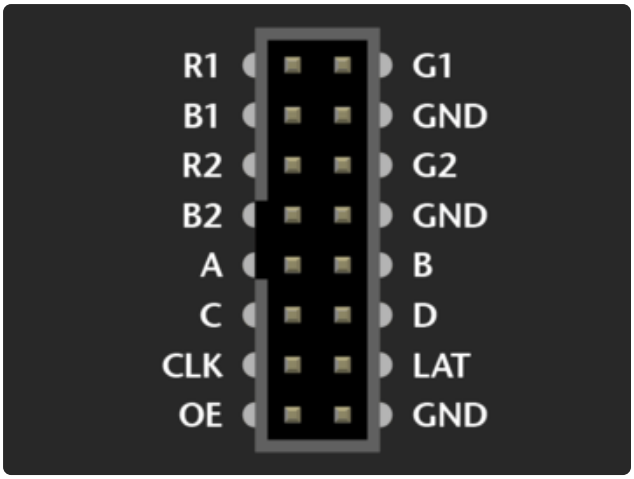
keys point the same way and the plugs face the same way, pins are in the same positions at both ends.



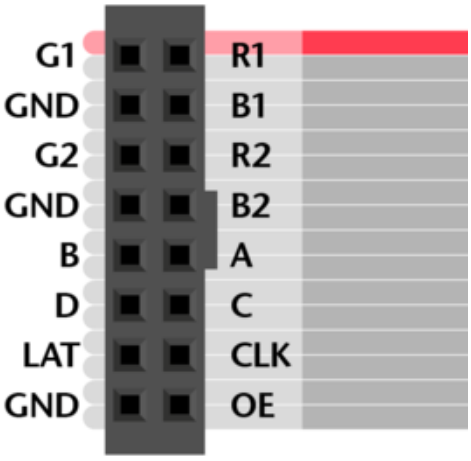
Plugged into a socket on the LED matrix, one header now faces away from you. If you double the cable back on itself (not a twist, but a fold)...to access a specific pin on the socket, the left and right columns are now mirrored (rows are in the same order — the red stripe provides a point of reference). You're looking "up" into the plug rather than "down" into the socket.

For example, R1 (the top-left pin on the INPUT socket) appears at the top-right of the exposed plug. You can jam a wire jumper in that hole to a corresponding pin on the Arduino...





So! From the prior page, refer to the socket that's correct for your matrix type. The labels may be a little different (or none at all), but most are pretty close to what's shown here.

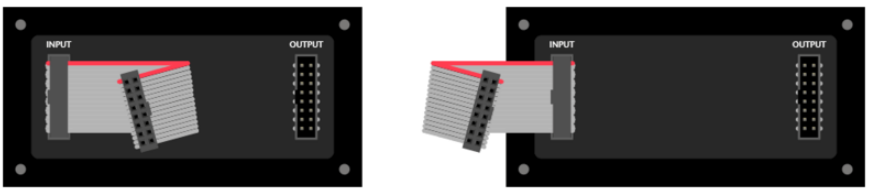


Then swap the columns to find the correct position for a given signal.

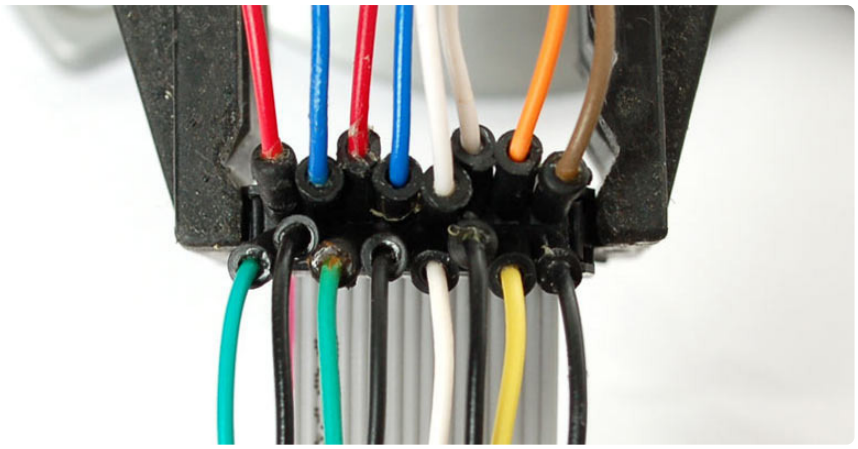
Either end of the ribbon cable can be plugged into the matrix INPUT socket. Notice below, the "key" faces the same way regardless.

With the free end of the ribbon toward the center of the matrix, the Arduino can be hidden behind it.

With the free end of the ribbon off the side, it's easier to see both the front of the matrix and the Arduino simultaneously, for making additional connections or for troubleshooting.

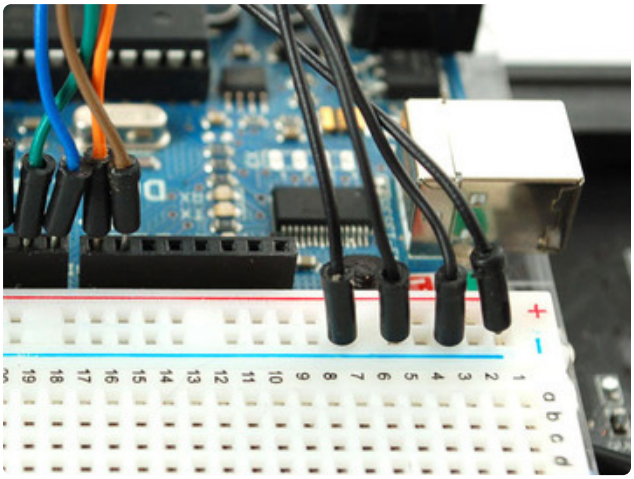


Using color-coded wires helps a lot! If you don't have colored wires, that's okay, just pay close attention where everything goes. Our goal is a fully-populated plug like this:

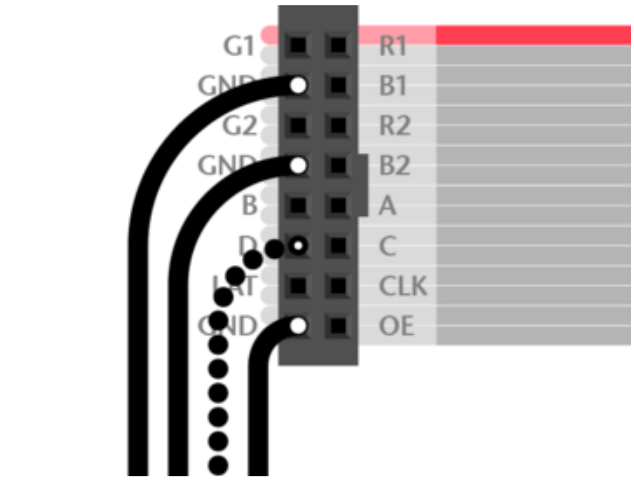


So! Let’s proceed with the wiring, in groups...

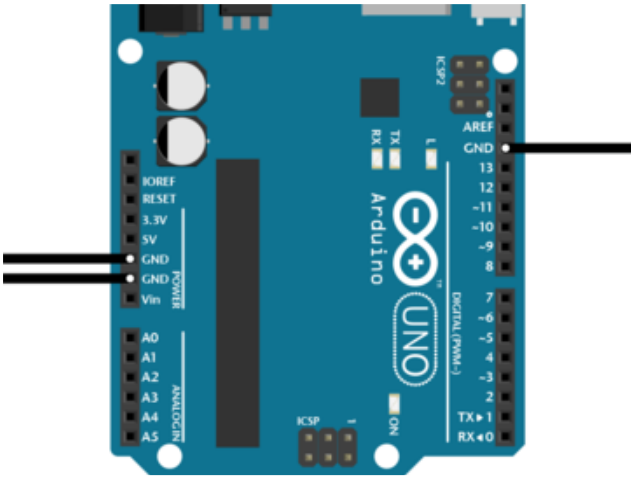
Connect Ground Wires



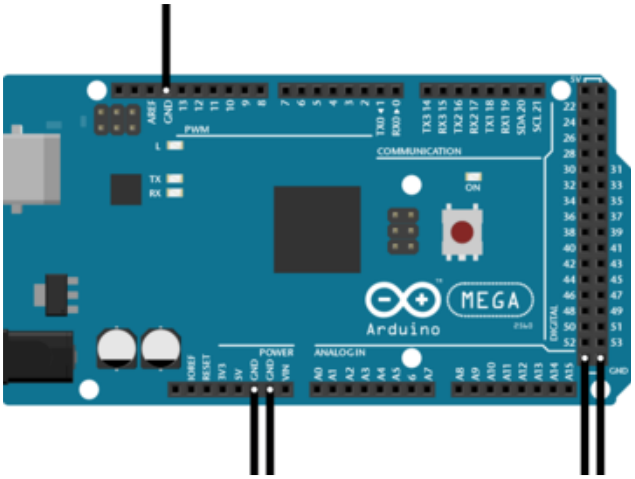
32x32 and **64x32** matrices require **three** ground connections. **32x16** matrices have **four**.



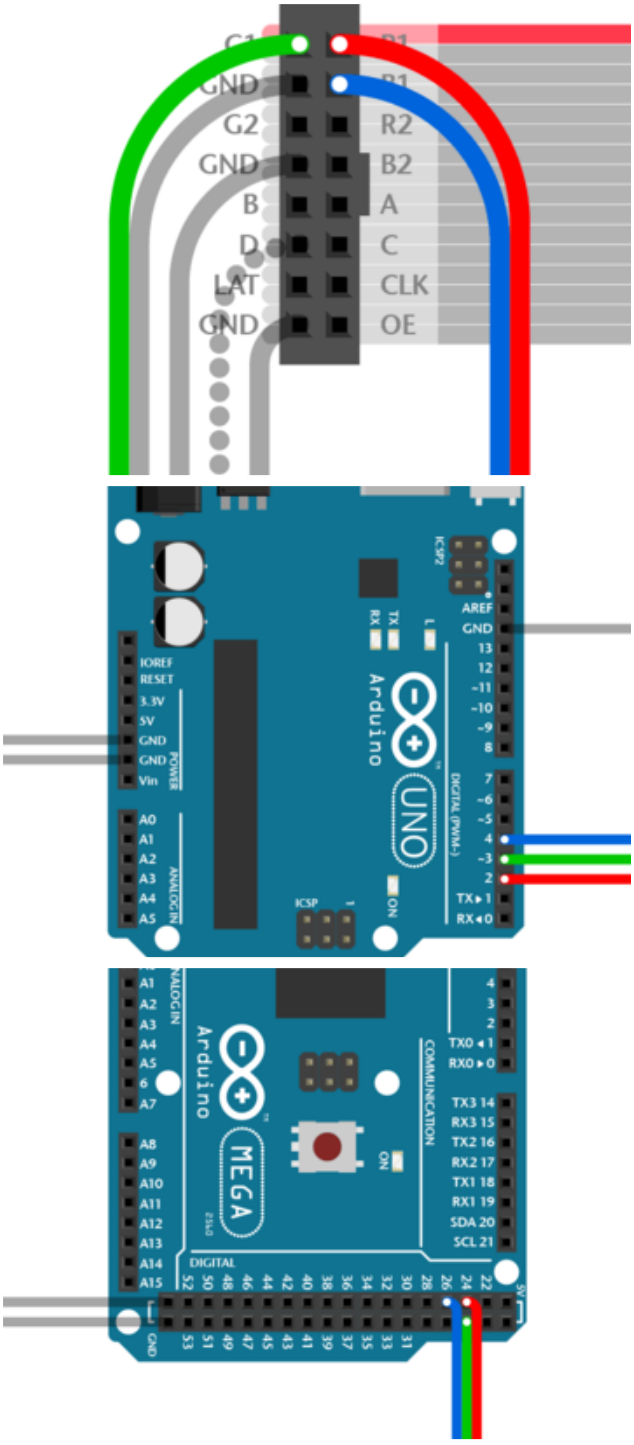
Current **Arduino Uno form-factor** boards have **three** ground pins (the third is next to pin 13). If you need additional ground connections — for a 32x16 matrix, or if using an older Arduino board with only 2 ground pins — a solderless breadboard is handy for linking all these pins.



Arduino Mega boards have **five** ground pins. Same three as the Arduino Uno, plus two more next to pins 52 & 53.


PWM
COMMUNICATION
ON
TX
RX
MEGA
Arduino
POWER
ANALOG IN
DIGITAL

Upper RGB Data

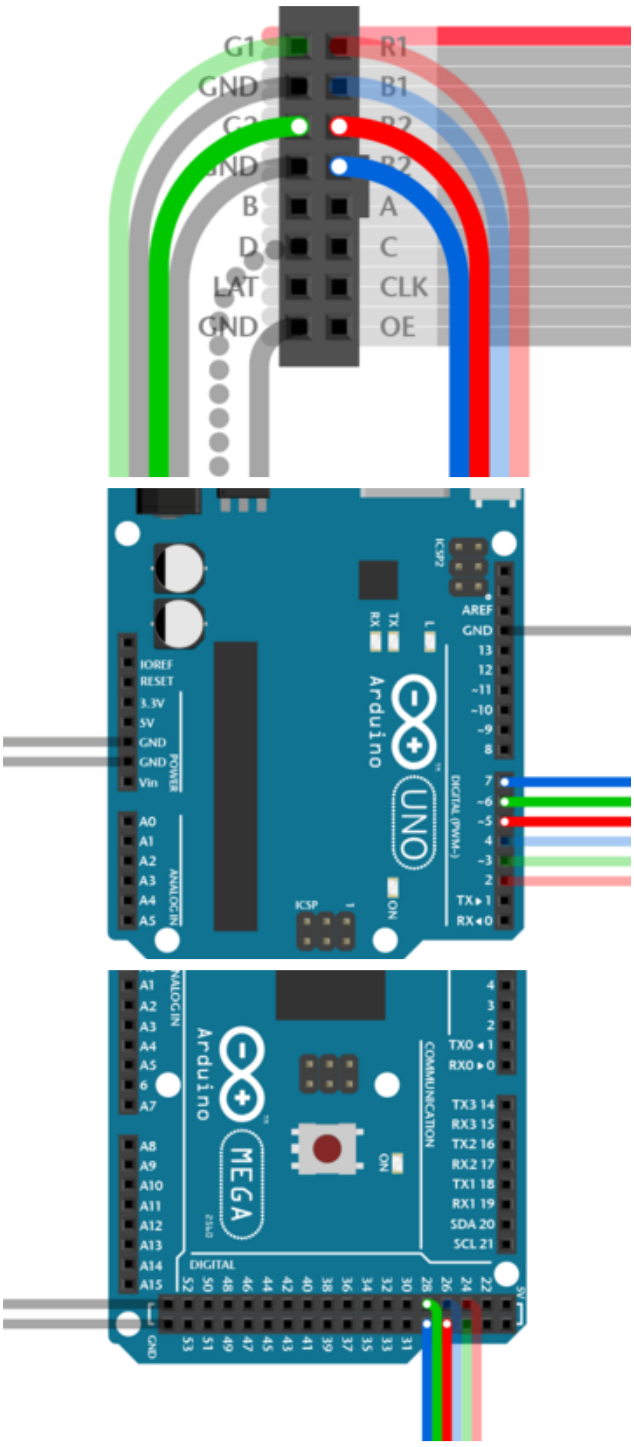


Pins **R1**, **G1** and **B1** (labeled R0, B0 and G0 on some matrices) deliver data to the **top half** of the display.

On the **Arduino Uno** and **Adafruit Metro (328, M0** or **M4)** boards, connect these to digital pins **2**, **3** and **4**.

On **Arduino Mega**, connect to pins **24**, **25** and **26**.

Lower RGB Data

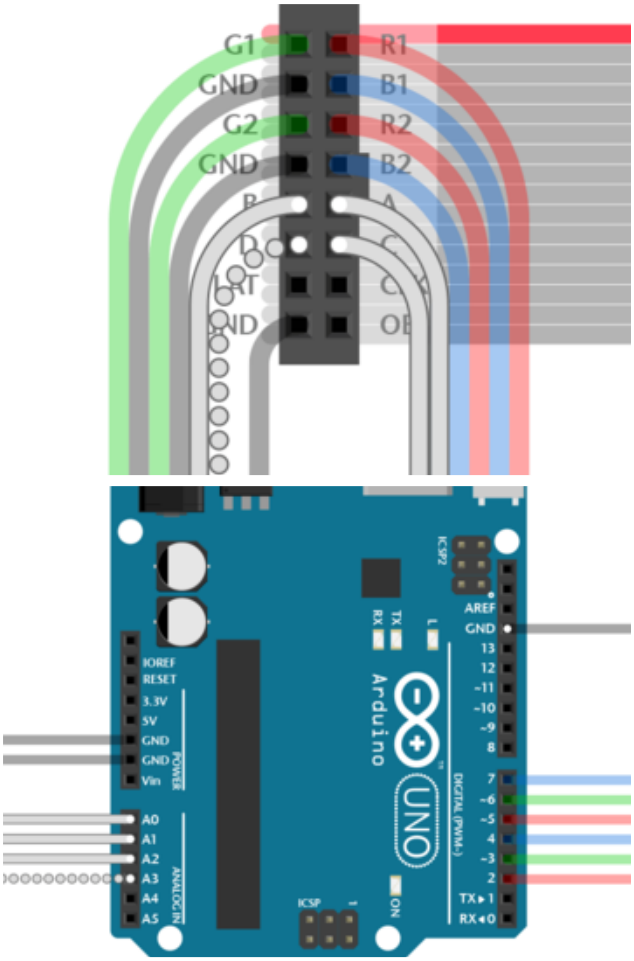


Pins **R2**, **G2** and **B2** (labeled R1, G1 and B1 on some matrices) deliver data to the **bottom half** of the display. These connect to the next three Arduino pins...

On **Arduino Uno** and **Adafruit Metros**, that's pins **5**, **6** and **7**.

On **Arduino Mega**, pins **27**, **28** and **29**.

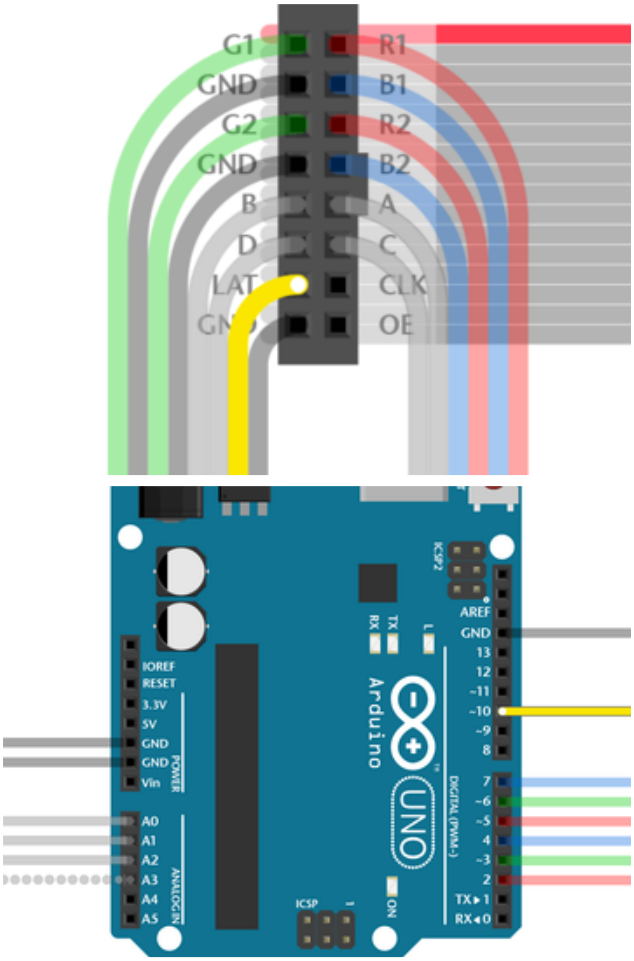
Row Select Lines



Pins **A**, **B**, **C** and **D** select which two rows of the display are currently lit. (**32x16 matrices don't have a "D" pin** — it's connected to **ground** instead.)

These connect to pins **A0**, **A1**, **A2** and (if D pin present) **A3**. This is the **same** for all boards.

LAT Wire

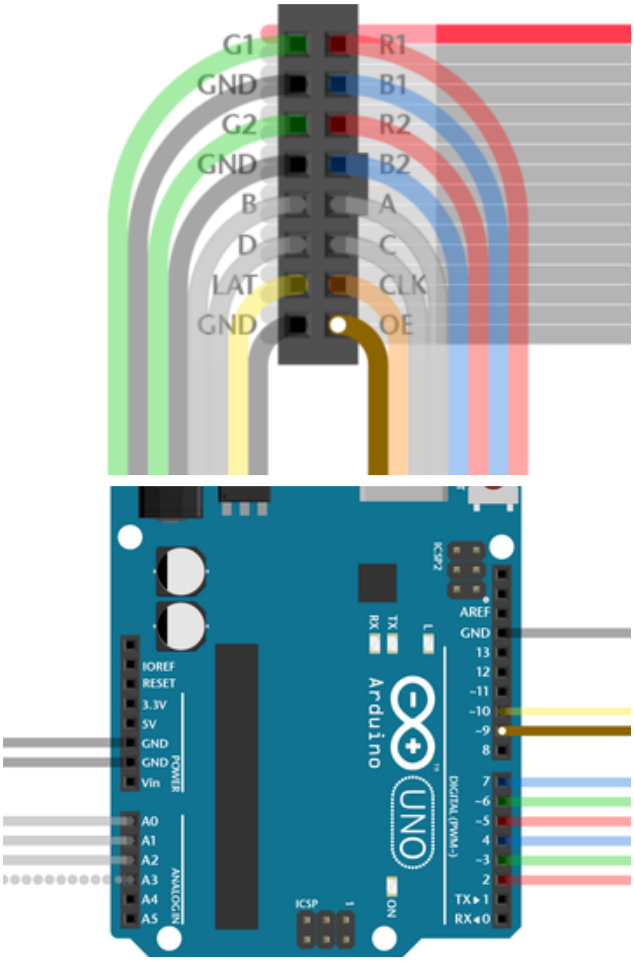


The **LAT** signal connects to Arduino pin **10**.

This is the same for all boards.

The LAT (latch) signal marks the end of a row of data.

OE Wire

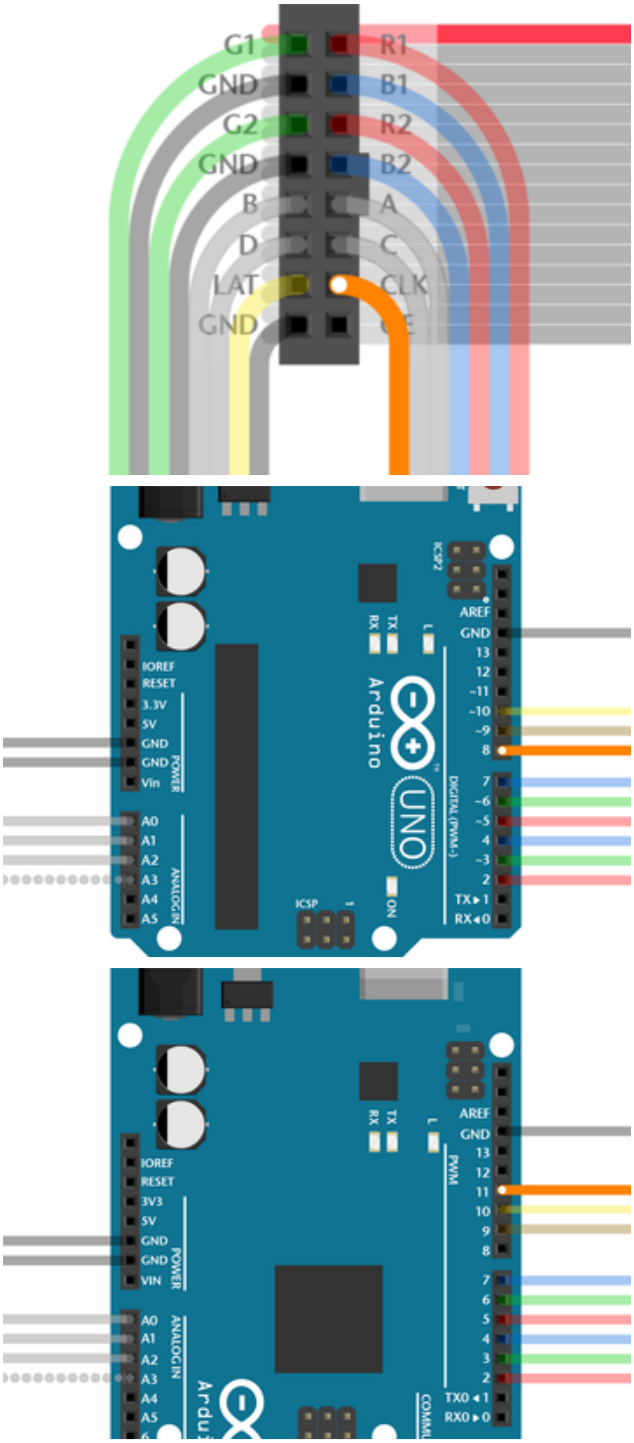


OE connects to Arduino pin **9**. This is the same for all boards.

OE (output enable) switches the LEDs off when transitioning from one row to the next.

CLK Wire

Last one!



CLK connects to...

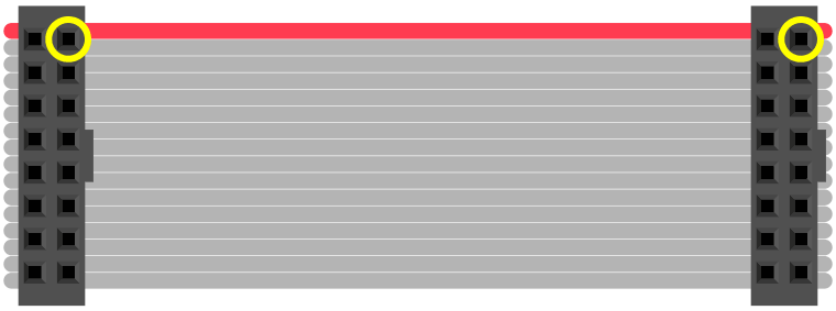
Pin 8 on **Arduino Uno**, **Adafruit Metro 328** or **Metro M0**.
Pin 11 on **Arduino Mega**.
Pin A4 on **Adafruit Metro M4** (not shown, but you get the idea).
The CLK (clock) signal marks the arrival of each bit of data.

That's it. You can skip ahead to the "Test Example Code" page now.

Connecting Using a Proto Shield

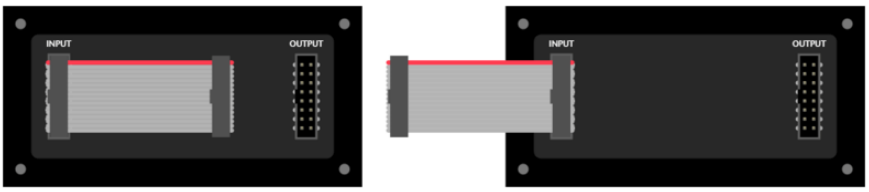
As mentioned on the "Jumper" page: if you hold a ribbon cable flat — no folds — and with both connectors facing you, keys pointed the same direction — there's is a 1:1 correlation between the pins. The top-right pin on one plug links to the top-right on the other plug, and so forth. This holds true even if the cable has a doubled-over

strain relief. **As long as the keys point the same way and the plugs face the same way, pins are in the same positions at both ends.**

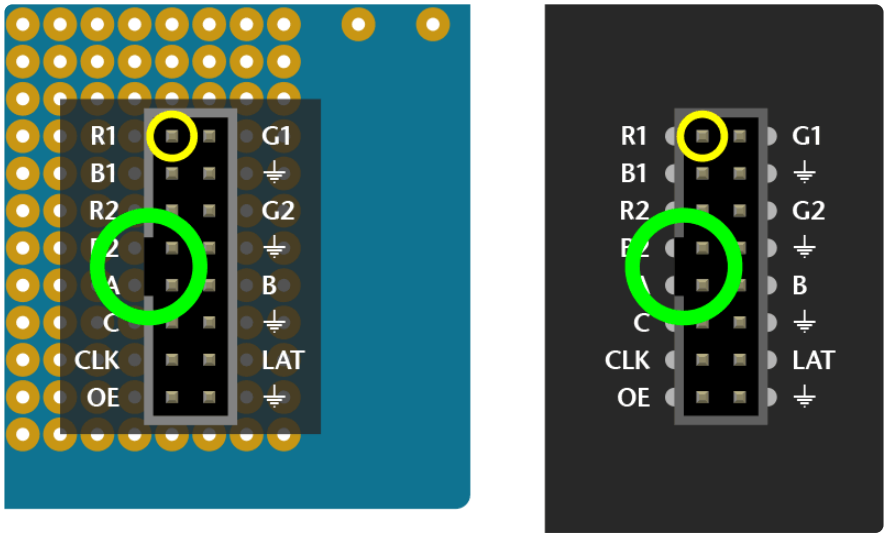


Either end of the ribbon cable can be plugged into the matrix INPUT socket.

The free end of the ribbon can point toward the center of the matrix, or hang off the side...the pinout is still the same. Notice below the direction of the "key" doesn't change.

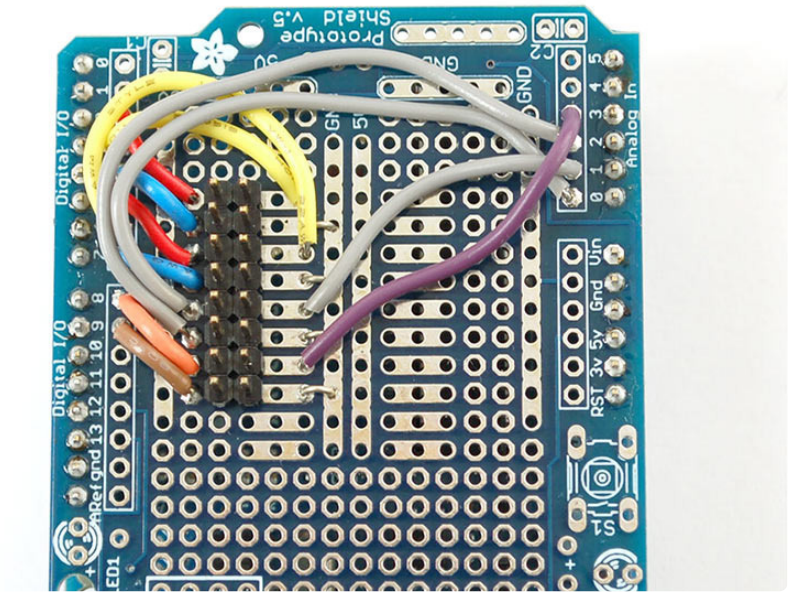


A dual-row header gets installed on the proto shield, similar to the connector on the matrix. Just like the ribbon cable lying flat, as long as these two headers are aligned the same way, they'll **match pin-for-pin**; unlike the jumper wire method from the prior page, mirroring doesn't happen.



Wires are then soldered from the header to specific Arduino pins on the proto shield. Try to keep wire lengths reasonably short to avoid signal interference.

Using color-coded wires helps a lot! If you don't have colored wires, that's okay, just pay close attention where everything goes. Our goal is a proto shield something like this:

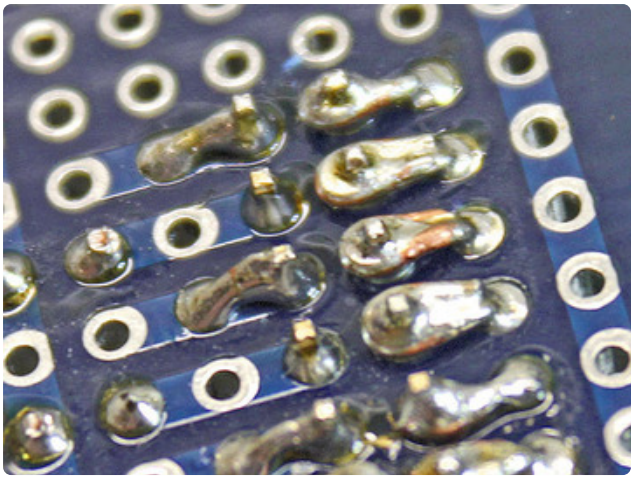


It's not necessary to install all the buttons and lights on the proto shield if you don't want — just the basic header pins are sufficient.

For **Arduino form-factor boards**, using an **Adafruit** proto shield (http://adafru.it/2077): **if using a shrouded socket (like on the back of the matrix — with the notch so a ribbon cable only fits one way) you'll need to place this near the "Reset" end of the shield.** The plastic shroud obscures a lot of pins. **Others' proto shields may be laid out different**...look around for a good location before committing to solder.

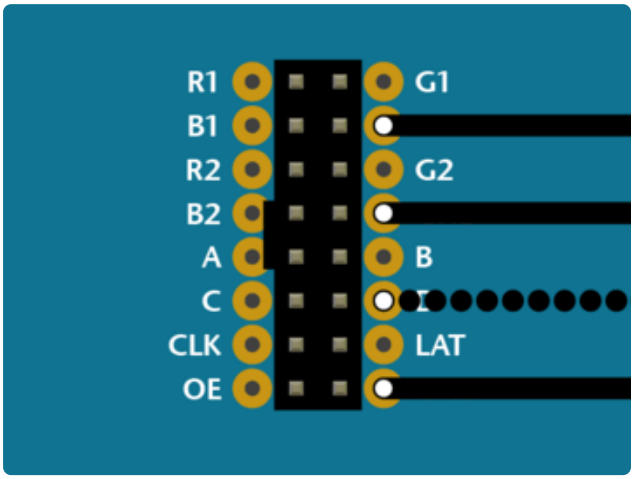
For **Arduino Mega** with our corresponding proto shield (http://adafru.it/192): a shrouded socket fits best near the **middle** of the shield.

Otherwise, you can use a plain 2x8-pin male header, or two 1x8 sections installed side-by-side (as in the photo above). Since there's no alignment key with this setup, you might want to indicate it with some tape or a permanent marker.



Depending on the make and model of proto shield, some pins are designed to connect in short rows. Others don't. For the latter, strip a little extra insulation and bend the wire to wrap around the leg of the socket from behind, then solder.

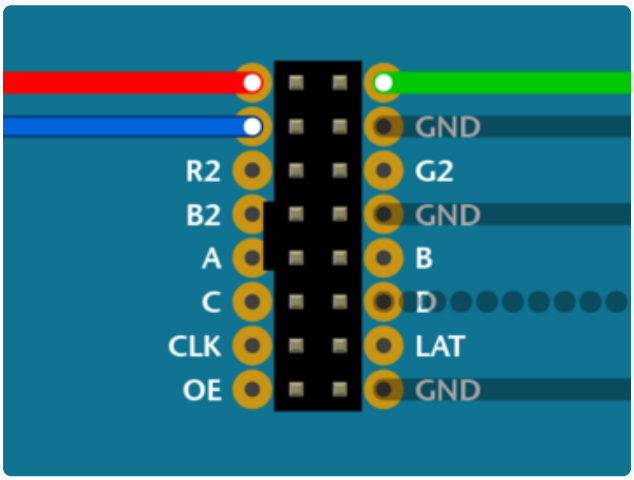
Connect Ground Wires



32x32 and **64x32** matrices require **three** ground connections. **32x16** matrices have **four**.

Most proto shields have tons of grounding points, so you shouldn't have trouble finding places to connect these.

Upper RGB Data

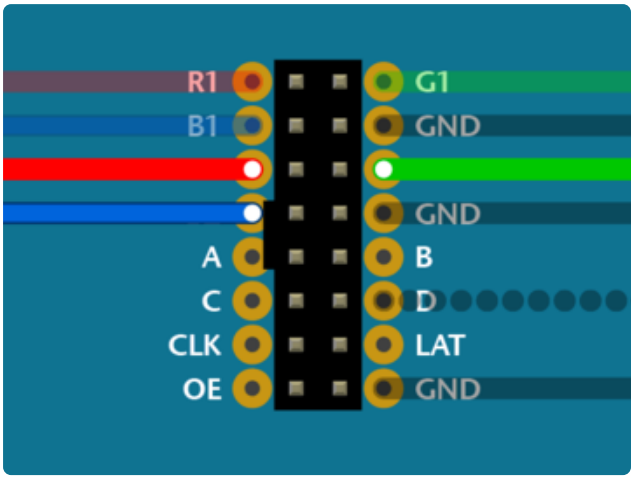


Pins **R1**, **G1** and **B1** (labeled R0, B0 and G0 on some matrices) deliver data to the **top half** of the display.

On the **Arduino Uno** and **Adafruit Metro (328, M0** or **M4)** boards, connect these to digital pins **2**, **3** and **4**.

On **Arduino Mega**, connect to pins **24**, **25** and **26**.

Lower RGB Data

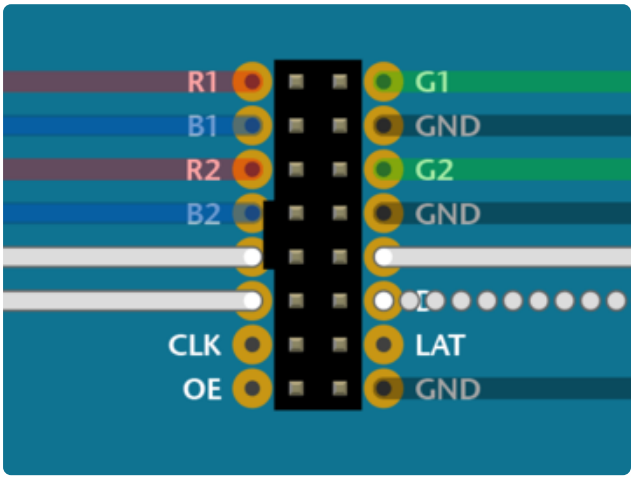


Pins **R2**, **G2** and **B2** (labeled R1, G1 and B1 on some matrices) deliver data to the **bottom half** of the display. These connect to the next three Arduino pins...

On **Arduino Uno** and **Adafruit Metros**, that's pins **5**, **6** and **7**.

On **Arduino Mega**, pins **27**, **28** and **29**.

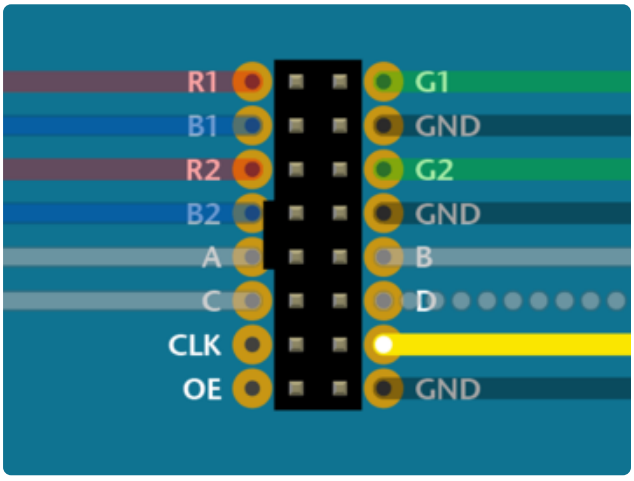
Row Select Lines



Pins **A**, **B**, **C** and **D** select which two rows of the display are currently lit. (**32x16 matrices don't have a "D" pin** — it's connected to **ground** instead.)

These connect to pins **A0**, **A1**, **A2** and (if D pin present) **A3**. This is the **same** for both the **Arduino Uno** and **Mega**.

LAT Wire

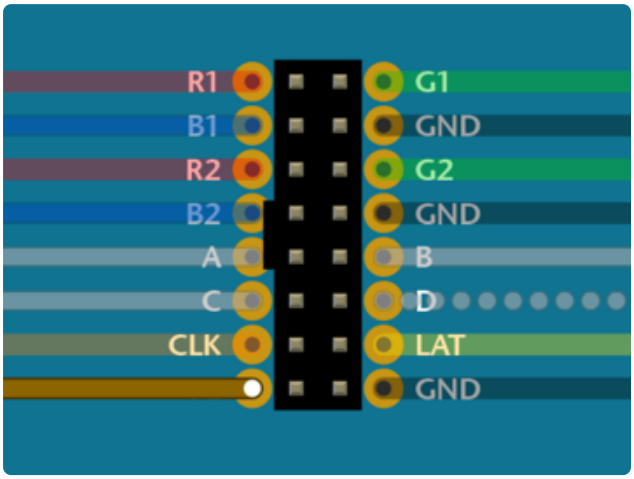


For **32x32** and **64x32** matrices, **LAT** connects to Arduino pin **10**.

This is the same for all boards.

The LAT (latch) signal marks the end of a row of data.

OE Wire

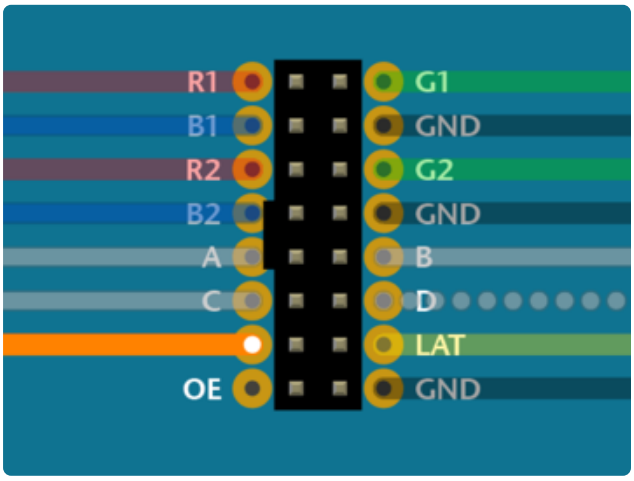


OE connects to Arduino pin **9**. This is the same for all boards.

OE (output enable) switches the LEDs off when transitioning from one row to the next.

CLK Wire

Last one!



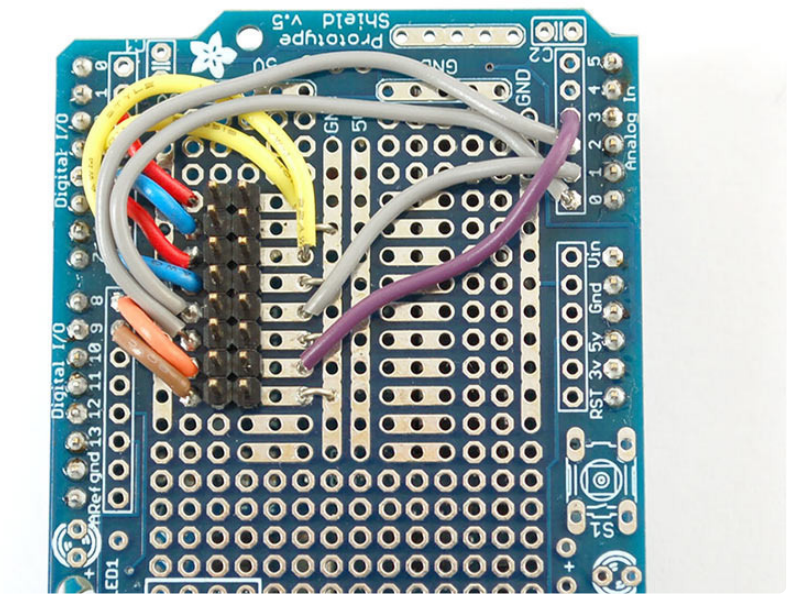
CLK connects to...

Pin 8 on **Arduino Uno**, **Adafruit Metro 328** or **Metro M0**.
Pin A4 on **Adafruit Metro M4**.
Pin 11 on **Arduino Mega**.
The CLK (clock) signal marks the arrival of each bit of data.

Here's that photo again of a completed shield. You can tell this is for a 32x16 matrix, because there are four ground connections (one of the long vertical strips is a ground bus — see the tiny jumpers there?).

The ribbon cable to the matrix would plug into this with the key facing left.

The colors and positions don't quite match the examples above, but are close. G1 and G2 are yellow wires. LAT is the purple wire and should go to pin 10 now (we changed around some things in the Arduino library).



Test Example Code

We have example code ready to go for these displays. It does not work with every board out there. **See the "Compatible Hardware" lists on the opening page for guidance.**

The library works with a LIMITED NUMBER of boards, including the Arduino Uno and Arduino Mega. The Arduino Leonardo, despite its shape, is NOT supported due to its peculiar pinout.

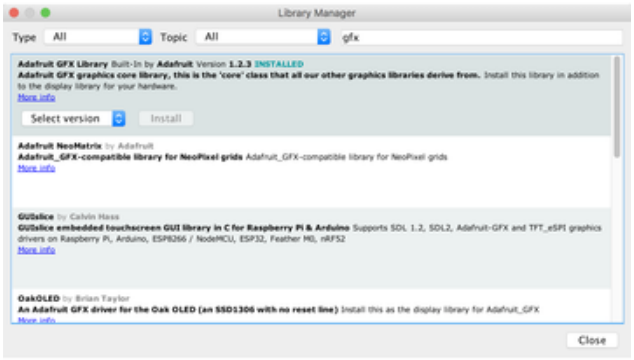
For 32-bit microcontrollers such as SAMD21 ("M0"), SAMD51 ("M4"), RP2040 and ESP32, please use the Adafruit_Protomatter library instead, which can be installed using the Arduino Library Manager.

Support for 32-bit microcontrollers — M4, RP2040, ESP32, etc. — is documented in a different library called Protomatter, documented here (https://adafru.it/MNa)**.**

Arduino Uno support is limited to the 32x16 matrix, and only "single-buffered" (i.e. no smooth animation), despite any comments in the examples that might suggest otherwise.

> Over time, RAM required by the core Arduino code and the matrix-driving graphics libraries has increased...a few bytes here and there as bugs are fixed and features are added. Early on, 32x32 (or double-buffered 32x16) just barely fit in the Arduino Uno's RAM, with a few dozen bytes to spare for user code. This is no longer the case. But you might still see references to this in older code.

Some libraries need to be downloaded and installed: first is the RGB Matrix Panel library (https://adafru.it/aHj) (this contains the low-level code specific to this device), then the Adafruit GFX Library (https://adafru.it/aJa) (which handles graphics operations common to many displays we carry) and Adafruit BusIO (https://adafru.it/GxD).





Both libraries can be found and installed using the Arduino Library Manager (**Sketch→Include Library→Manage Libraries...**). Search for “gfx” and “rgb matrix panel” and install the corresponding Adafruit libraries.

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). There’s also a **testshapes_32x64** example for boards with sufficient RAM.

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:

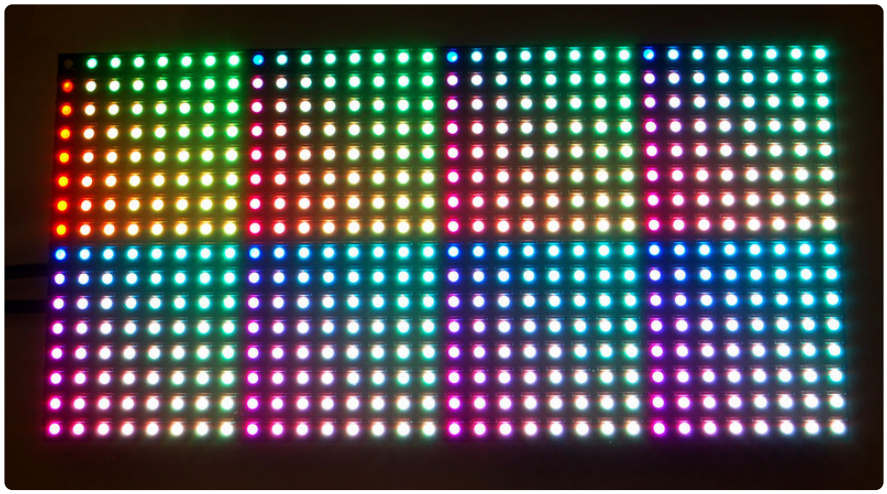
```
#define CLK 11
```

(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.)

If using an **Adafruit Metro M4** (not M0 or 328), the CLK change would instead be:

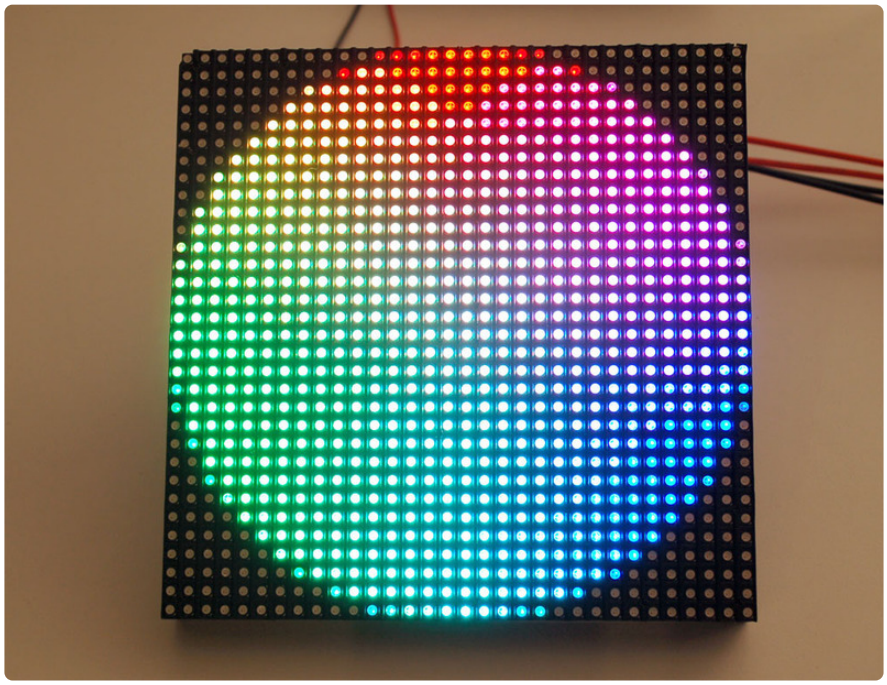
```
#define CLK A4
```

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:

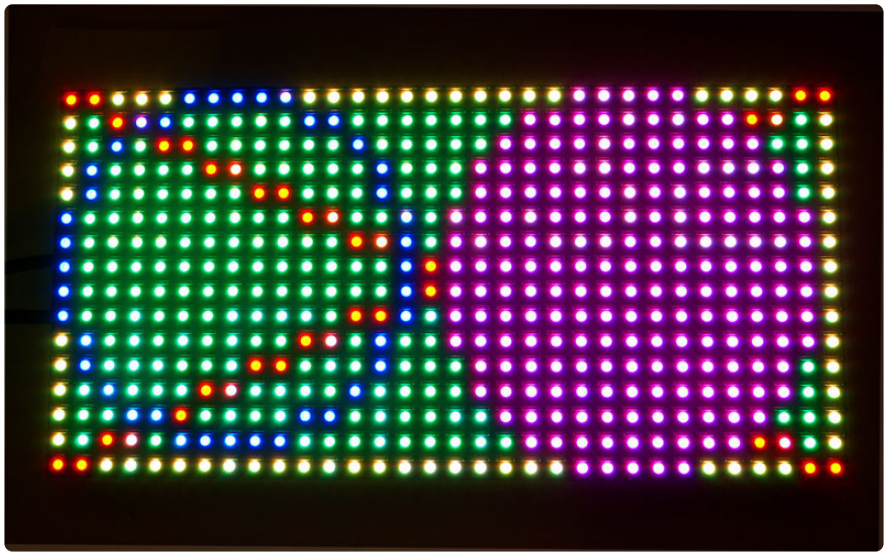
```
      matrix.drawPixel(x, y, matrix.Color333(r, g, b));
```

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 funciton **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.

RGBMatrix 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));
```

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

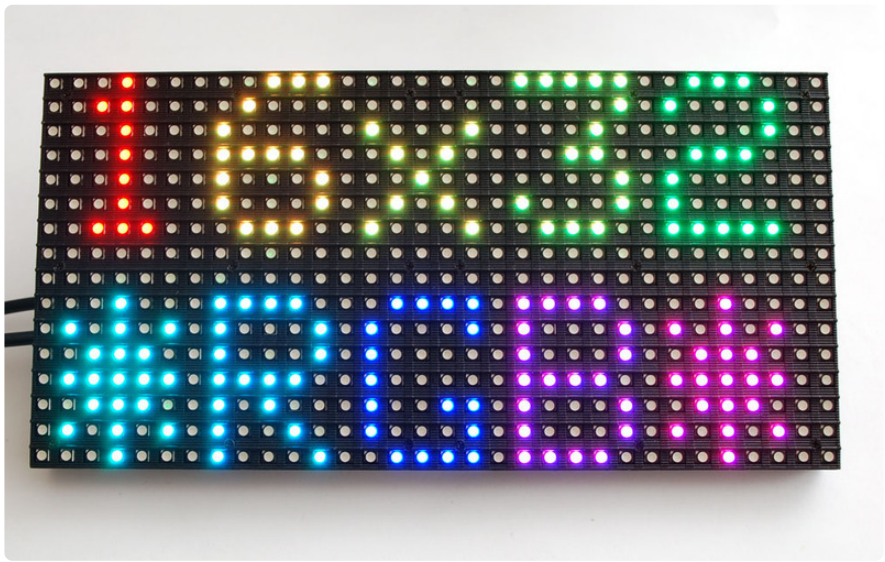
```
// fill the screen with 'black'
  matrix.fillScreen(matrix.Color333(0, 0, 0));
```

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("*");
```



How the Matrix Works

There's no “official” documention 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 8th and 16th line. You might be asking, why are the lines paired this way? wouldnt 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 are 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 (https://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:

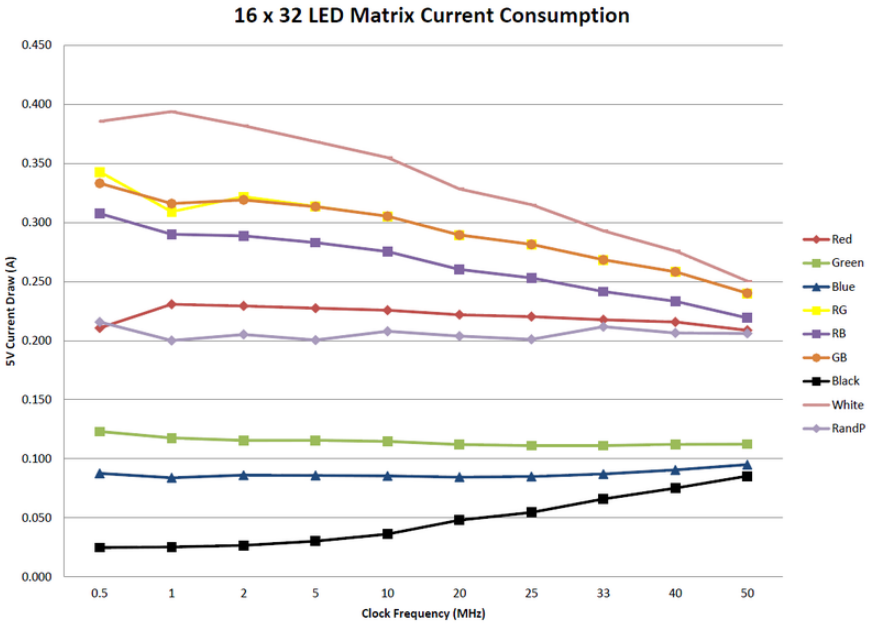


Image above by rhb.me (CC By-NC-SA)

“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.”

FAQ

I'm seeing weird pixel artifacts, ghosting, or other glitches.

This will probably be due to one of these:

- Wiring
- Power
- Software
- Bad display

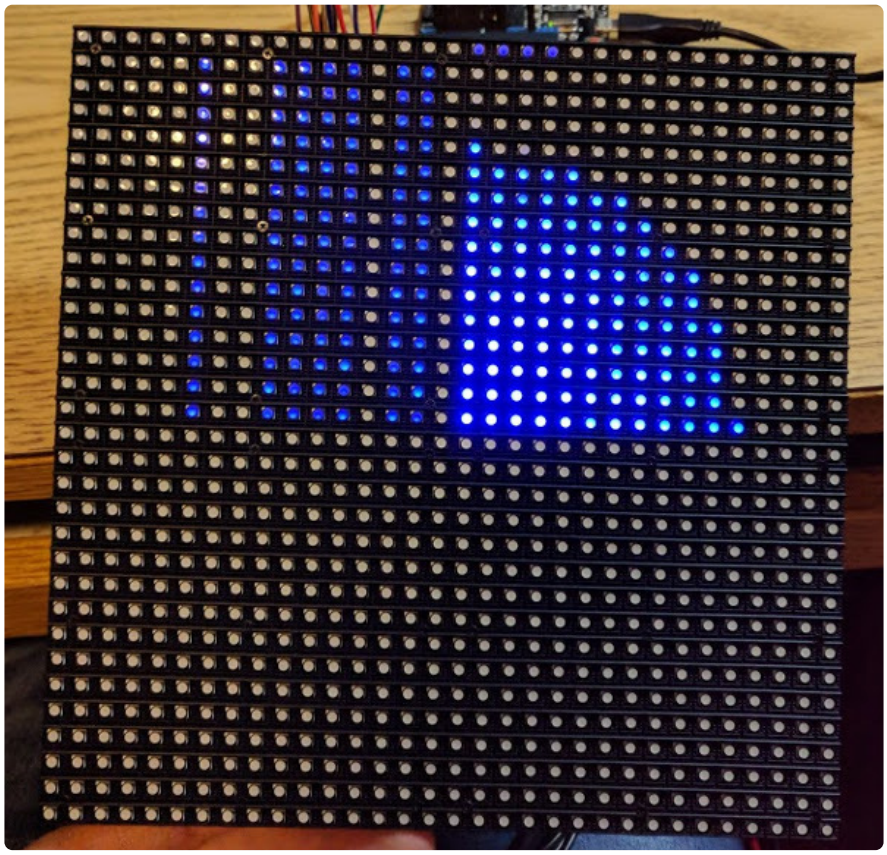
Wiring problems are probably the biggest source of display issues. Especially if you are attempting the "jumper wire" connection method. There have also been cases of power supplies causing interference. This guide covers both proper wiring and power.

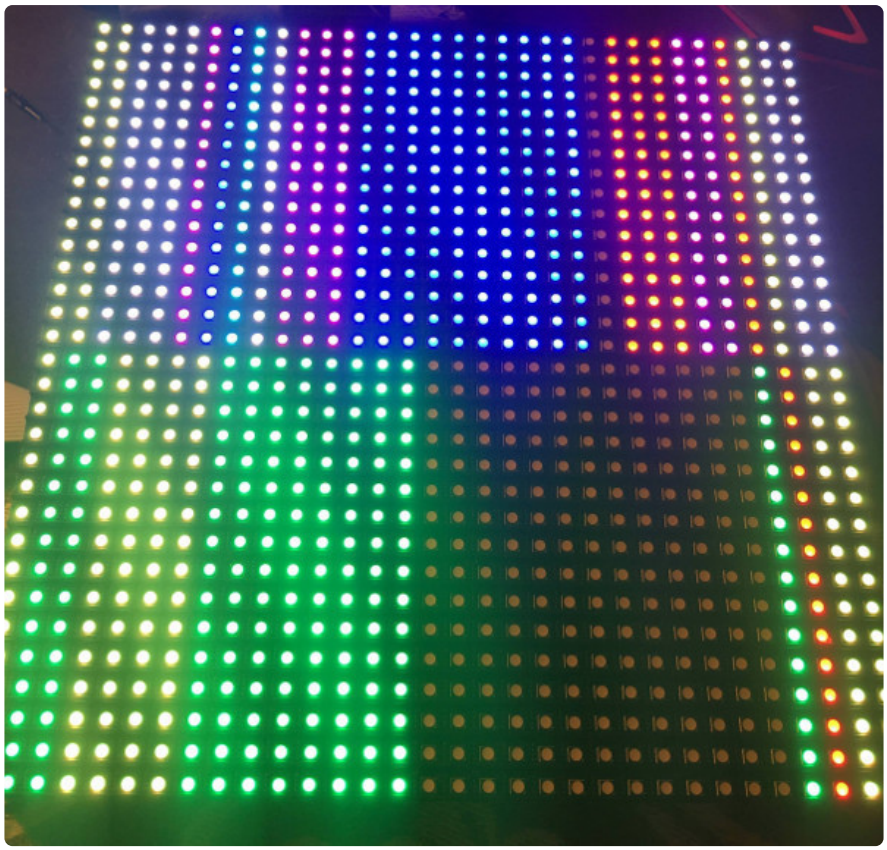
If all of that seems OK and you are still seeing weird pixels, it may be due to how the pixels are being driven in software. You can try some tweaks as covered in the guide, but the issue may not entirely go away.

If you've tried to troubleshoot and still have issues, post in the forums and we can take a look. Every now and then a display can just be bad.

Wiring Issue Examples

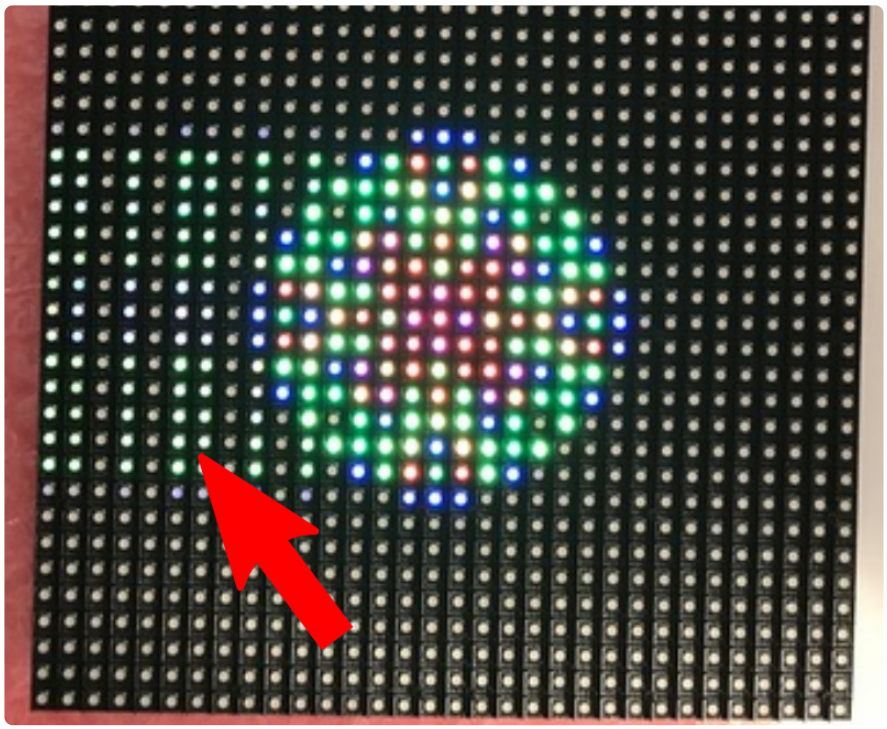
Wiring issues tend to look like this and are often fixable if you can find the wrong / bad wire.

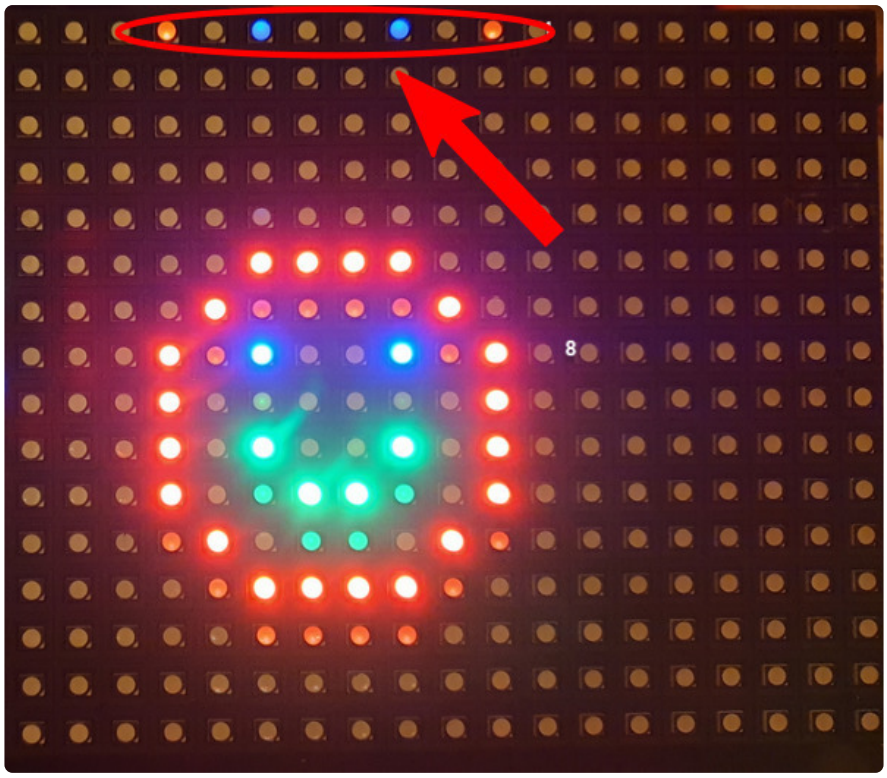




Ghosting Issue Examples

You see what you expect for the most part, but also some other artifacts. This can happen even with proper wiring.



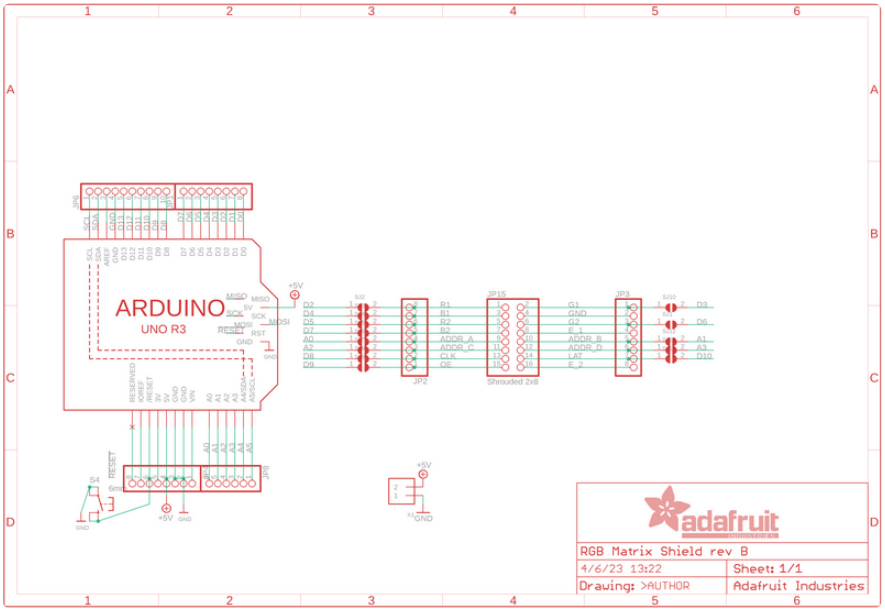


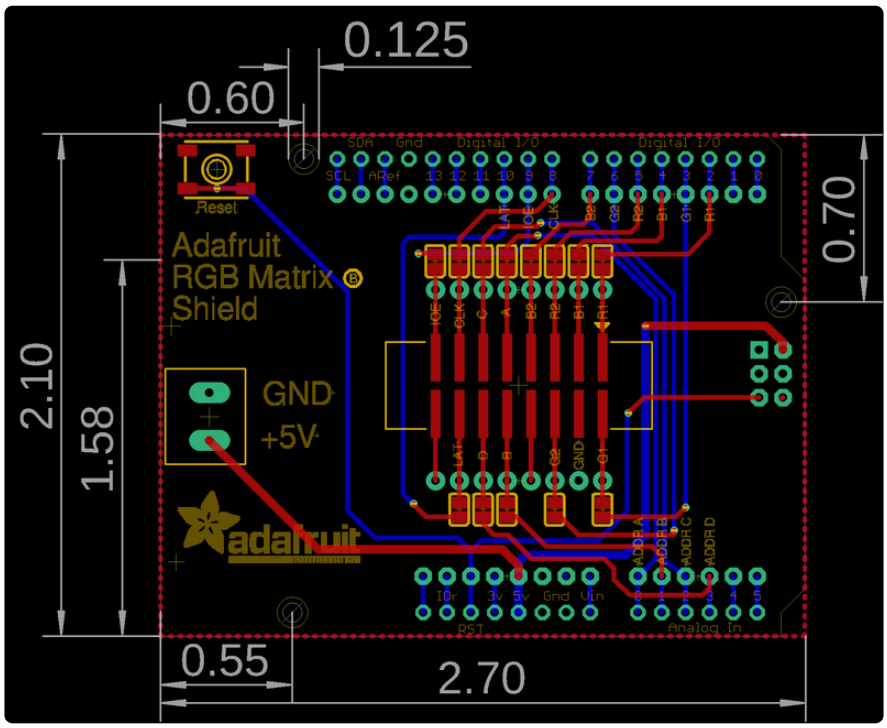
Downloads

Files

- Fritzing object in the Adafruit Fritzing Library (https://adafru.it/aP3)
- EagleCAD PCB Files in GitHub (https://adafru.it/BNu)

Schematic and Fab Print for Partially Assembled Version





Schematic & Fabrication Print Original Version

