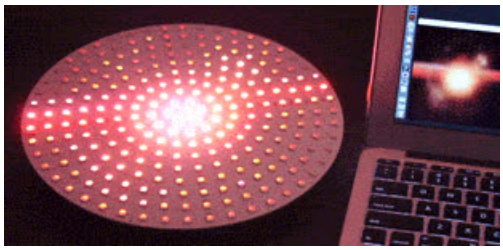
adafruit learning system

LIGHTSHIP: LED Animation over WiFi

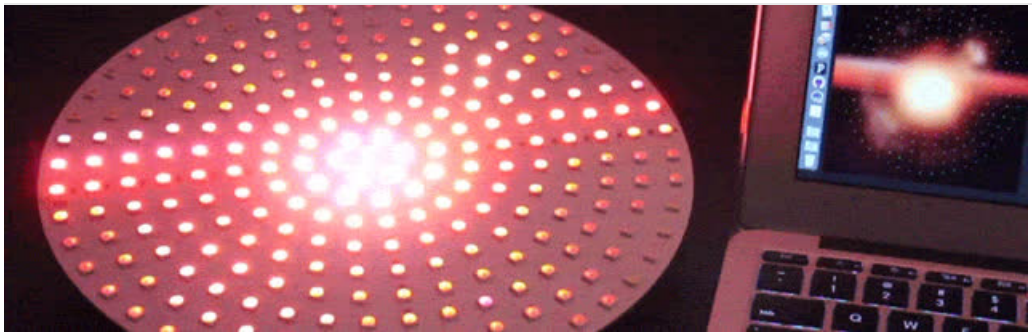
Created by Phillip Burgess



Last updated on 2018-08-22 03:52:09 PM UTC

Guide Contents

Overview



***Open Pixel Control* (OPC)** is a protocol for driving arrays of RGB lights. Unlike *DMX,* which is tied to specific cabling, voltages and topology, OPC leverages existing interconnects like Ethernet or even WiFi. OPC is particularly well-suited to **LED art installations!**

Creating mobile, *self-contained* OPC displays — processor, wireless networking, LEDs and battery in a single robust unit that one can carry — has been an ongoing challenge. In this guide we'll create one such device, using the **Adafruit Feather M0 WiFi** to replace multiple elements with a single small package.

In the past, this might've been handled with a combination of parts, such as a Raspberry Pi computer with a USB WiFi adapter and *Fadecandy (http://adafru.it/1689)* controller (a USB-based OPC device for NeoPixels). This works, but the Pi board uses more power, the long delay for Linux to boot is an annoyance, and the multitude of separate parts raises durability concerns...USB cables can pop out, SD cards are easily jostled and corrupted in such mobile environments. Still other approaches have relied on custom firmware for portable WiFi routers.

There are quite a few hardware and software components in this project, and some prior programming experience is assumed...please **read through the whole guide first to understand all the pieces before committing.**

So is this like a replacement for Fadecandy?

Far from it! We implemented some FadeCandy-like ideas, such as dithering and interpolation, and it is can use the same FadeCandy software, but the nature of the hardware may require dialing back the frame rate or number of LEDs. Ours is, at best, a **modest approximation.** It's *fun-size* 'Candy.

High-end installations will still benefit from the Raspberry Pi + Fadecandy duo. The Pi keeps up with fast datastreams, while the Fadecandy hardware fully exploits its potent *M4* processor, with firmware written by two of the brightest minds in embedded development: Micah Scott and Paul Stoffregen.

You're using DotStar LEDs. Can I use NeoPixels instead?

No. We found the Feather M0's processor better equipped to handle a single long DotStar strand over high-speed SPI using DMA...we effectively get that time "free." That's not an option with NeoPixels on this particular board - while you can DMA a strand of NeoPixels you can't do 8 in parallel. Perhaps we'll revisit this idea with various hardware in the future.

Can I use an ESP8266 instead of an ATWINC-equipped board?

No. The code for this project exploits specific hardware features of the Feather M0 processor...it won't just copy over and run on the ESP8266. But again, maybe we'll explore other options in the future.

Other hardware options?

It can also work with an **Arduino Zero** and **WiFi Shield 101**. It's not as compact and you won't get USB battery

charging...but if you already have the parts around, this lets you prototype and experiment with the idea; you can move it over to the smaller hardware later.

Hardware

You'll have some options for batteries and other parts...more on that in a moment...but the **board-to-DotStar connections are always the same:**

Feather M0 or Arduino Pin #	**DotStar**
11	DI (Data In)
13	CI (Clock In)
GND	GND or –

Make sure you're connecting the board to the *input* end of the DotStar strip or matrix. It should be labeled **DI** and **CI.** Don't connect the board to DO and CO...that's the *output* end.

As written, our Arduino code can support up to 512 DotStar LEDs. This is to mimic Fadecandy's 512 pixel support, so existing examples for that device might carry over. However...

- The LED topology will be different. This project uses a *single long DotStar chain* vs. Fadecandy's 8 concurrent NeoPixel chains.
- Our hardware isn't as powerful as Fadecandy...with 512 LEDs, you'll probably need to reduce the frame rate to 30 FPS or less (we'll cover this later). But shorter runs work fine with fast frame rates.

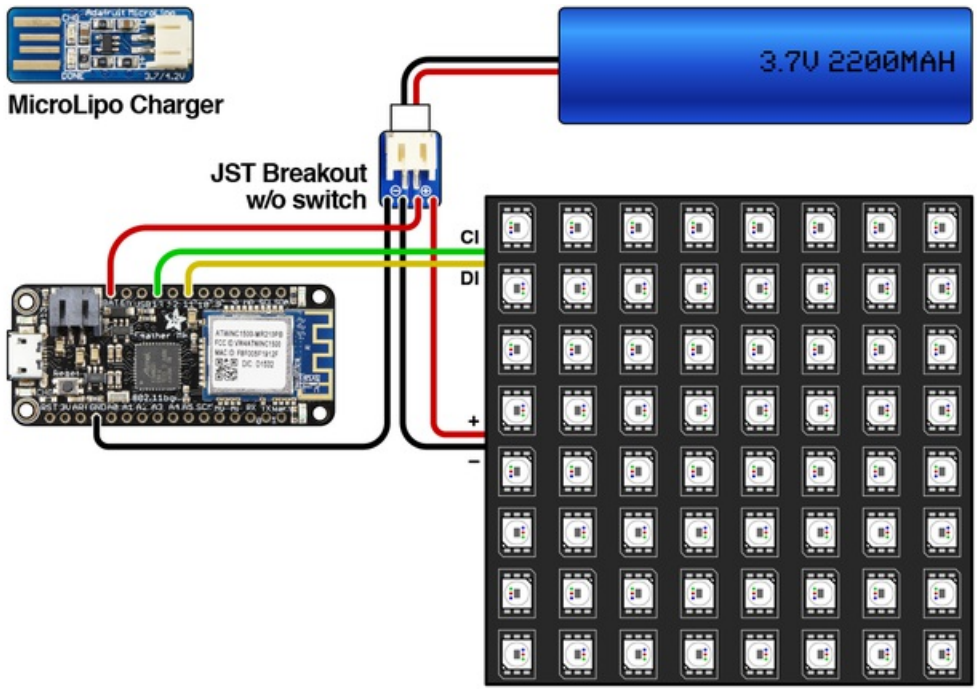
Powering large numbers of LEDs in portable projects also presents challenges. So...though we *can* handle 512 LEDs, for practical reasons you'll most likely run fewer than that. **One or two meters of DotStar strip** should work well, or an **8x8 matrix** (maybe 16x16 with a large enough battery).

We'll illustrate with some 8x8 DotStar matrices, but the real power of this system is that it can take *any* shape, whatever you might assemble from sections of DotStar strip.

Battery Power

The Feather M0 board has a battery charging circuit built in. It works great for circuits using sensors and small displays...but DotStars use a *lot* of power, and batteries for this project will tend to be large. So for practical reasons, you're best served foregoing the built-in charger. Physically disconnect the battery when not in use, and use a dedicated LiPoly charger (such as our MicroLipo USB charger) to top it off. (If the battery's over 500 mAh, bridge the pads on the back of the MicroLipo with a dot of solder to enable 500 mA charge rate.)

Power from the battery needs to be split to both the DotStars and the Feather M0 board. Use a JST socket breakout board (*don't* use the version with a switch, it's not rated for high current), or you can hack one end off a JST extension cable and split power two ways.

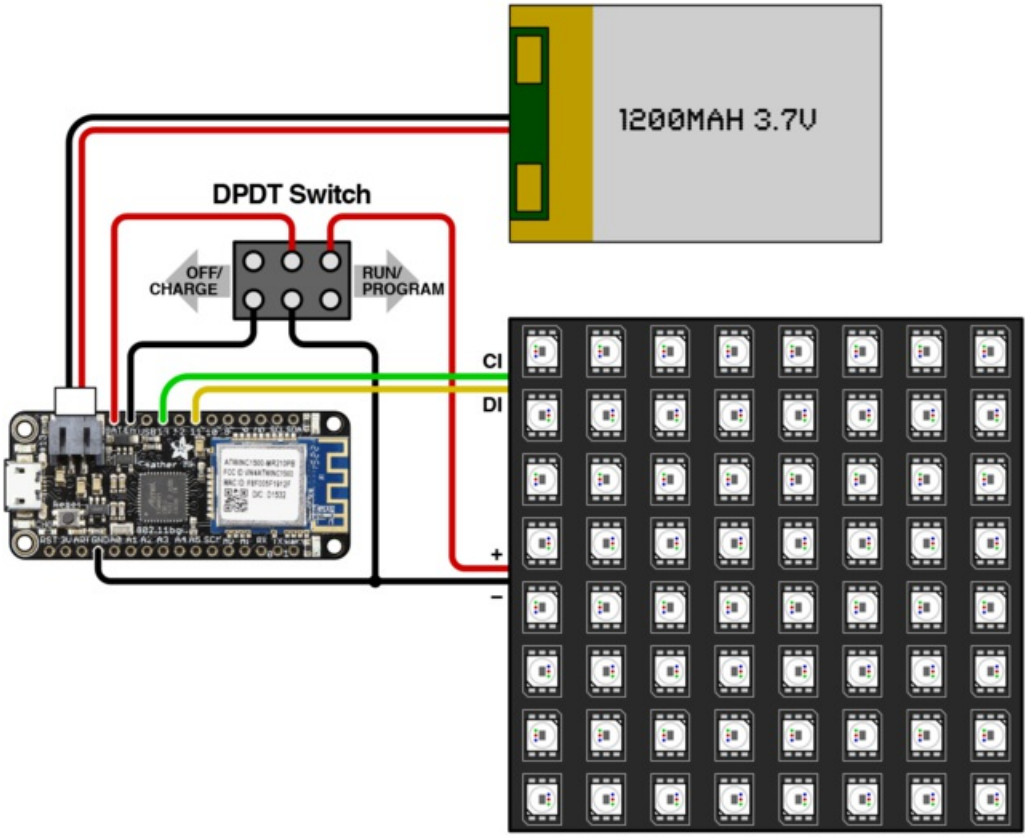


Battery	**DotStars**	**Feather M0**	or **Arduino Zero**
+	5V	BAT	Vin
–	GND	GND	GND

With smaller projects, the Feather M0's built-in charger becomes more practical. It provides about a 200 mA charge… so it depends on the battery size and your patience whether this is worthwhile…500 to 1200 mAh seems reasonable. Also it requires one extra part which **isn't sold by Adafruit:** a double-pole, double-throw (**DPDT**) switch. You can order something from DigiKey or might find one at Radio Shack if you still have one nearby. Just make sure that the switch you use is rated for at least a couple Amps of DC current.

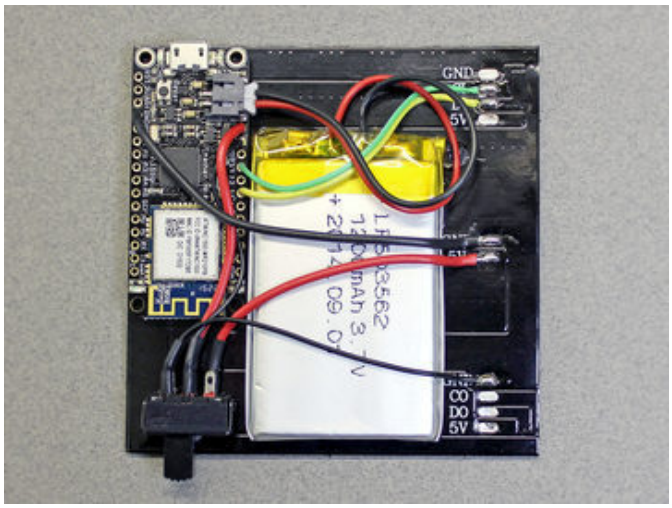
Follow the diagram below for switch wiring; only 4 of the 6 pins on the switch are used. When in the "off" position, you can connect a USB cable to charge the battery. In the "on" position, the Arduino sketch runs, or you can upload new code to the board.

Don't use this scheme for large projects…the Feather M0 can pass through about 1 Amp of current, with occasional higher surges. With lots of LEDs, you'll do better with the first circuit layout.



In tests, the 1200 mAh cell works well with the 8x8 matrix (and should also accommodate an equivalent amount of DotStar strip). Scaling up gets challenging…you could try a 16x16 matrix with a 2500 mAh battery (direct, as in the first circuit, *not* using the Feather as a pass-through), but that's *4X* the pixels with only about 2X the battery capacity. For large installations, you may want to step up to a 4400 (http://adafru.it/354) or 6600 (http://adafru.it/353) mAh lithium ion battery pack…or, if *really* large, add a DC jack and plug into a 5V wall supply, though this sacrifices portability.

Alternately…plan your animations to limit the overall brightness and the number of LEDs lit at any one time.



Here's a small widget built with an 8x8 matrix and 1200 mAh LiPoly cell, along with an old DPDT switch I had on hand. As a quick prototype, this just uses foam tape and hot glue…but for something more durable and presentable I'd be inclined to make a 3D-printed enclosure.

Look closely and you'll see the switch connects the board's En (enable) pin to one of the unused GND pads on the matrix when in the "off" position. There's continuity between all three GND pads, so I exploited this to simplify the soldering (rather than the three-way splice shown in the diagram). Get creative!

Software

If this is your first time using the Adafruit Feather M0 board...intermission time! Read our Feather M0 Basic Proto guide to set this up:

Adafruit Feather M0 Basic Proto — Arduino IDE Setup(https://adafru.it/ldF)

Newer versions of the Arduino IDE have been released since that guide...1.6.5 and 1.6.7 both seem to work pretty well with the M0 board.

If you *have* used the Feather M0 before, use the Arduino IDE Boards Manager to check that you're using the latest *Adafruit SAMD Boards* support files. (For the Arduino Zero, install the *Arduino SAMD Boards* package.)

Before continuing, make sure you can compile and upload sketches to the board. Try the basic "Blink" example as a test.

To provide WiFi support and for our code to use certain M0-specific features, it's necessary to download three libraries. The first of these — **WiFi101** — can be found using the **Arduino Library Manager** interface (Sketch→Include Library→Manage Libraries...). The other two libraries are new and experimental, so they're not in the Library Manager, you'll need download and install these manually in your Documents/Arduino/Libraries folder:

https://adafru.it/lnc

https://adafru.it/lnd

Then download and extract the ZIP file containing the code for this project:

https://adafru.it/ldG

In this archive are two folders:

The first, "**Arduino**," contains the OPCserver sketch for the Feather board (or Arduino Zero).

The second, "**Processing**," contains several Open Pixel Control client demos for use with the *Processing* programming language. These will run on your main computer (desktop or laptop).

The client/server nomenclature may seem odd...the OPC "server" runs on the tiny Feather board, while OPC "clients" are programs running on a larger and more capable system. All the color and animation decisions are made in the client applications...the server just passes these through to the LEDs.

OPC clients can be written in many programming languages, but we'll use ***Processing*** (a derivative of Java) as it's free, cross-platform (runs on Windows, Mac and Linux) and is focused on visual arts.

Processing Downloads Page (https://adafru.it/cK1)

For now, we recommend downloading the **version 2.2.1 release** of Processing for your operating system. The 3.X series introduced some significant changes that aren't always compatible with existing Processing code.

Processing looks a *lot* like the Arduino IDE (in fact, the Arduino IDE derived from the same code base). This can be confusing because Arduino sketches don't work in Processing, nor vice versa. Make sure you're loading sketches into the correct IDE for each.

Edit Arduino Sketch

In the Arduino IDE, open the "OPCserver" sketch. We'll make some changes before uploading to the board, to configure for your particular hardware and network.

Toward the top of the code, the following lines are of interest. They're not in one place, but all appear in the first 50 or so lines of the sketch:

```
#define ADAFRUIT_ATWINC
```

The above line tells the code to work the Adafruit WiFi hardware. If using an Arduino Zero w/WiFi Shield 101 and Native USB, comment it out. Next, look for...

```
//#define Serial SerialUSB // Enable if using Arduino Zero 'Native USB' port
```

The above is commented out by default. Enable this line *only* if you're using an Arduino Zero *and* are connected to the "Native USB" port (rather than the "Programming" port). Then...

```
#define IP_TYPE IP_STATIC // IP_STATIC | IP_DYNAMIC | IP_BONJOUR
```

You'll probably leave this line as-is, which tells the WiFi module to use a static IP address (rather than dynamically assigning an address from your WiFi router). A static address makes it easier for OPC clients to access the device, as it's always in a known location.

This requires some knowledge of how your WiFi router doles out IP addresses. Most will have a numeric range of IP addresses to assign dynamically...for example, my WiFi router starts issuing dynamic IP addresses at 192.168.0.100 and above. I can assign fixed addresses below that (except for 0 and 1) to specific devices, as long as others aren't using them.

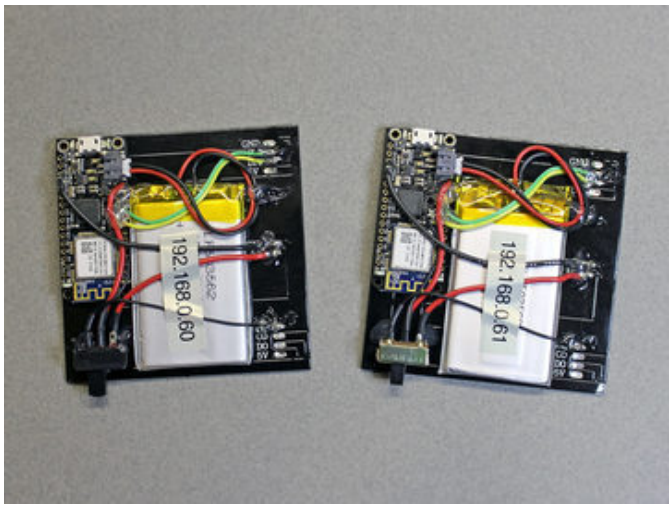
```
char       ssid[] = "NETWORK_NAME",   // WiFi credentials
           pass[] = "NETWORK_PASSWORD";
```

Replace the above two strings with the login credentials for your wireless network. Finally...

```
IPAddress  ipaddr(192, 168, 0, 60); // Static IP address, if so configured
```

This is the static IP address to be assigned to the device on your wireless network. As mentioned above, this requires knowledge of your router's DHCP policy. Most will start with 192.168.0.X, 192.168.1.X, 10.0.0.X or 10.1.1.X — but not all.

Whatever address you use, you'll need this later for the Processing sketches.



If making multiples, assign each one a unique IP address. Label them to help keep track!

A little further down in the code (around line 100) are these lines:

```
#define DOTSTAR_BLUEBYTE  0
#define DOTSTAR_GREENBYTE 1
#define DOTSTAR_REDBYTE   2
```

You probably don't need to edit these. But later, if you find the test examples are producing the wrong colors, you may need to return here and edit these numbers. This is the “native color order” used by the DotStar LEDs...it’s changed at least once in different production runs of these devices. If your LEDs are from a different batch, you’ll need to swap or rearrange some of these numbers (they’ll always be the values 0, 1 and 2, only the order changes).

Upload

Make sure **Adafruit Feather M0** (or **Arduino Zero** if using that board) is selected in the Tools→Board, then upload the code to the board.

If the code **doesn’t compile** (throws an error):

- Confirm the right board type and USB port are selected in the Tools menu. Have you installed the correct files for this board using the Boards Manager?
- Confirm the correct libraries are installed: **Adafruit_ASFcore**, **Adafruit_ZeroDMA** and **Adafruit_WINC1500** (for Feather M0) or **WiFi101** (for Arduino Zero + WiFi Shield 101).
- There may be problems if *both* the WiFi101 and Adafruit_WINC1500 libraries are installed. If so, remove the WiFi101 library...the WiFi Shield 101 works fine with the WINC1500 library, you’ll just need to edit some pin numbers (this is commented in the code).
- Check that you haven’t mangled the syntax on any of the lines edited above. For example...the IP address has commas (not periods) between each value.

If the code **compiles but doesn’t upload**:

- If you built the circuit with the DPDT switch, this needs to be in the “RUN” position to upload code.
- Try uploading again. Or tap reset once or twice on the board, then upload. The M0 boards are a new thing and

can be a bit persnickety what with all the different operating systems and USB port types.

Now open the Arduino IDE Serial Monitor.

If you see nothing at all, that's actually a good sign. Or if you see a "Server listening" message, that's good too.

If you get an unending series of periods (...), the board isn't connecting to your wireless network. Confirm that the network name, password and provided IP address are all valid.

If everything checks out, let's try the examples...

OPC Client Apps

Hello world! Launch the Processing (2.2.1) IDE and load the first of our example programs: **OPCstrandtest.**

At the top of the code are these two lines:

```
OPC opc       = new OPC(this, "192.168.0.60", 7890);
int numPixels = 256; // Set this to actual strand length
```

The first line holds the address of your OPC server device. We previously configured that in the Arduino code. The format here is just a little different though…instead of four comma-delimited values (192, 168, 0, 60), here it's a string with period separators "192.168.0.60". Edit the numbers to match the Arduino sketch.

Next line is the number of LEDs in your DotStar chain. This might be 64 for an 8x8 matrix, or 60 (or 144, etc.) for one meter of DotStar strip.

When you run this code (the top-left icon in the Processing window), after just a moment's delay you should see the message "Connected to OPC server" and get a "chaser" down the DotStar chain that cycles between red, green and blue for each pass. Press the ESC key to stop the program.

If the colors appear in the wrong order (not red, green, blue), you'll need to edit the Arduino sketch to match your particular DotStar hardware. This is explained near the bottom of the "Software" page.

If it runs slowly or stutters, especially with long LED runs, you may need to reduce the frame rate. Inside the setup() function you'll see a call to frameRate(). It takes a single argument, the number of frames per second. The default (if not specified) is 60 frames per second…that's usually fine with small projects, but with lots of LEDs you may need to dial it back, to perhaps 30 frames per second.

If you don't see the "Connected" message and don't get any LEDs: either the Server sketch on the Arduino can't connect to the wireless network, or the address at the top of the Processing sketch is incorrect.

Do not continue until you have the led "chaser" working, and it's cycling from red to green to blue.

Animation Without Programming

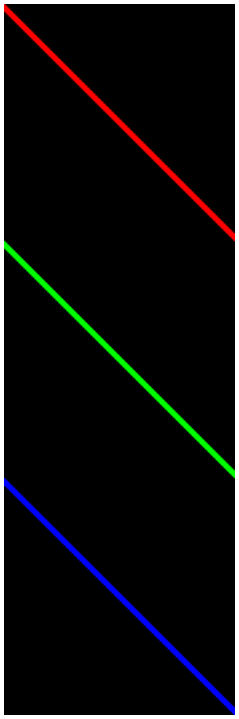
Now load the second example: **OPCpianoRoll**. Make similar edits to the IP address and LED chain length at the top of the code. When run, this will prompt you for an image file. A moving cross-section (one horizontal line) of this image is then "played" on the LED chain.

What this basically enables you to do is use your favorite image editor (Photoshop or most anything else) to create animation sequences. Create a new image whose pixel width is equal to the number of LEDs in the chain, and the height is the animation length in seconds times the number of frames per second (which can be controlled with the slider at the bottom of the window). For example: 60 DotStar LEDs, with a 15-second animation at 30 frames/second (the default) = 60 pixels wide x 450 pixels high (15 x 30). Make the background black, then try using the pencil or airbrush tools to paint some colorful lines. Save the result as a **lossless 24-bit image** (like PNG or TIFF…do not use GIF or JPEG, they'll degrade with each successive edit)…then run the OPCpianoRoll sketch and select this file when prompted.

The topology (a straight line) won't necessarily match the physical layout of your LEDs, but once you understand the "mapping" between the two different spaces it's pretty easy to use.

Here's an image equivalent to the RGB chaser code along a 144 pixel strip. Right click and "Save Image" to save this to

your computer for use with the OPCpianoRoll sketch:



Video Playback

Last example is “OPCvideo.” This works best with LED matrices. Edit the IP address at the top of the code to point to the OPC server, then edit the values of “arrayWidth” and “arrayHeight” to match the LED matrix (assumed to be a “zigzag” layout…if not, there’s an opportunity to change that a little further down in the setup() function). When run, you’ll be prompted to select a movie file (e.g. .MOV, .MP4, .AVI) which then plays on the LED matrix, hot damn.

Video decoding and playback is unfortunately not always reliable in Processing…it can be *incredibly* persnickety about codecs, plug-in libraries, 32-bit vs 64-bit and other things. If a certain video file doesn’t work, your only recourse may be to try other videos using different codecs, or sometimes a different version of Processing…or you may just have to skip this one and try the other demos (and start writing some of your own).

Some of the dimmer LEDs flicker!

This is normal and by design. DotStar LEDs have a finite number of discrete brightness levels, and our eyes perceive disproportionally large steps toward the low end of the brightness range. Borrowing an idea from Fadecandy, our code uses *temporal dithering* to approximate in-between brightness levels that would otherwise be too dim or too bright…it quickly alternates between the two. It’s less noticeable at a distance and/or with some diffusion over the LEDs, like paper or white acrylic.

Creating New OPC Clients in Processing

Notice that all of the example sketches have a second tab called “OPC.” This is an Open Pixel Control library for Processing, written by Micah Scott (Fadecandy’s creator). To create a new OPC sketch, or adapt an existing Processing sketch to add OPC output, make a new tab called “OPC” and then copy-and-paste that code from any of the existing OPC examples.

Processing is *way too deep* to summarize here. Entire books have been written on the subject. If you’d like to get started writing new sketches of your own, look through the Examples they include, or skim the Reference and Tutorials

pages on the Processing web site (https://adafru.it/ldl).

The OPC library allows you to create an “overlay” atop the screen output of any existing Processing sketch, redirecting specific pixels to an OPC server. Several functions allow you to position lines or grids (even circles) of LED pixel locations within the Processing output window. After this one-time setup, your sketch just draws animation as it would normally…no per-frame processing is required…and the LED pixels follow suit. So you can write animation code first (no hardware required), then worry about the LED placement later.

What’s exciting about this is that you’re *not constrained to pixel grids.* Your art can take whatever shape you feel it needs…this isn’t necessarily dictated by the physical hardware.

Here’s an example using one of our 255-pixel DotStar LED discs:

And the corresponding source code. Notice the “circumference” array, which lists the number of pixels in each concentric ring around the DotStar disc, and the calls to opc.ledRing() which overlay these points atop the program’s display window. The draw() function then handles the animation…it doesn’t have to think about networking or data packets at all, it just *happens.*

```
// OPC example scrolls text on a 255-pixel DotStar disc

OPC opc = new OPC(this, "192.168.0.60", 7890);
PFont f;

void setup() {
  size(400, 400);
  frameRate(60);

  int circumference[] = { 12*4, 11*4, 10*4, 8*4, 7*4, 6*4, 5*4, 3*4, 6, 1 };
  int i = 0;
  for(int j=0; j<10; j++) {
    opc.ledRing(i, circumference[j], width/2, height/2, width/20*(9-j), 0);
    i += circumference[j];
  }

printArray(PFont.list()); // Show available fonts, if you need to change:
  f = createFont("HelveticaNeue-BoldItalic", 450);
  textFont(f);
  textAlign(LEFT);
  colorMode(HSB, 100, 100, 100);
}

int hue = 0, x = -1800;

void draw() {
  background(0);
  fill(hue, 100, 50);
  text("Adafruit", x, 360);
  x -= 12;
  if(x < -1800) x = width;
  hue++;
  if(hue >= 100) hue = 0;
}
```

Not Done Yet

Things get *really* interesting when you realize it's possible to create *arrays* of OPC objects, each one pointed to a different remote OPC device...

```
OPC opc[] = {
  new OPC(this, "192.168.0.60", 7890),
  new OPC(this, "192.168.0.61", 7890),
  new OPC(this, "192.168.0.62", 7890)
};

void setup() {
  size(400, 400, P3D);
  opc[0].ledGrid8x8(0, 100, 200, 10, 0, true);
  opc[1].ledGrid8x8(0, 200, 200, 10, 0, true);
  opc[2].ledGrid8x8(0, 300, 200, 10, 0, true);
  ...
```

...once set up, the remainder is "automagic." The animation is synchronized across all the OPC devices on the wireless network, which are free to move around (within range of the WiFi router). This could make for some very potent and interesting performance art. Also, it's "self-healing"...if a device loses its connection, animation continues across the others, and the missing device will pick up at the right spot once a network connection is reestablished.

Python Too...and More to Come!

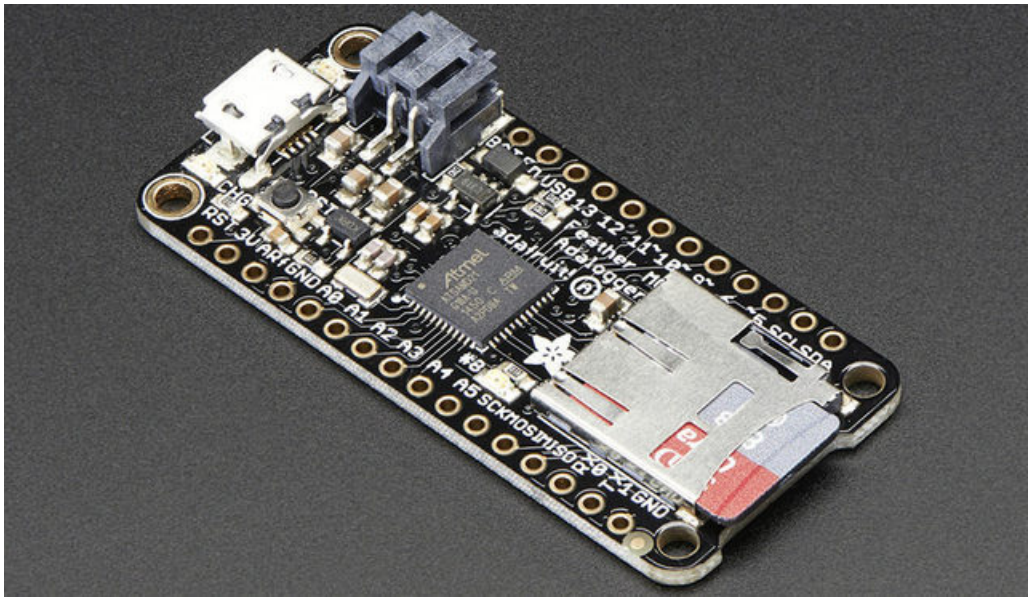
The Open Pixel Control protocol itself is fairly straightforward, and most any programming language with network support could conceivably be used to write clients.

The canonical OPC code repository on Github (https://adafru.it/ldJ) includes some examples written in Python.

Google around, you may find libraries for other languages as well. Or read the OPC stream specification (https://adafru.it/ldK) (incredibly simple!) and implement your own.

SD Card Playback

A variant of this project uses an **Adafruit Feather M0 Adalogger (http://adafru.it/2796)** (instead of M0 WiFi) to playback prerecorded animation sequences from a microSD card. This is great for environments where WiFi contention is an issue, or if you just don't want to have the Processing-side computer around. You won't get the cool multiple-synchronized-devices effect, but for a single device it's dandy.



The circuit is **identical,** just swap out the Adafruit M0 WiFi for an M0 Adalogger instead. Pinouts are the same, including the battery options (e.g. optional DPDT switch).

Processing Code Adjustments

The required code changes are minimal. Rather than creating an OPC object with a network address and port...

```
OPC opc = new OPC(this, "192.168.0.60", 7890);
```

...instead, specify an integer frame rate (frames per second) and an output file as a string (absolute paths are best):

```
OPC opc = new OPC(this, 30, "/Volumes/4GB/anim01.opc");
```

You can leave out the frames-per-second argument to use the default value of 30:

```
OPC opc = new OPC(this, "/Volumes/4GB/anim01.opc");
```

Avoid using the frameRate() method elsewhere in your code...the change won't be noted in the output file and playback will occur at a different rate. The OPC constructor sets your program's frame rate *and* records this in the file.

The Processing library doesn't much care about the output filename (as long as it's a valid location and has write permission)...but the Arduino code will scan the card for files with the extension "**.opc**", so it's recommended you use that.

There's one more step...recording to a file will not commence until you call the enable() method:

```
opc.enable();
```

This can be used to make sure recording does not actually begin until a video file is actually loaded...for example, look at the OPCvideo example sketch, notice enable() is called inside the movieEvent() method. This prevents several seconds of solid black being recorded to the file while the user navigates to and selects a video file. In other examples that don't require user input, enable() is called within setup()...recording can begin immediately.

When run, the Processing sketch will now output pixel data to this file (once the "enable()" method is called). It will continue until the program is terminated, either manually (e.g. Escape key) or through the exit() method.

On the Arduino Side...

In the Arduino IDE...instead of the OPCserver sketch, open **OPCstreamSD** from the same repository and upload that to the board.

OPCstreamSD will scan the root directory of an SD card; it does not look in subfolders. Any file ending in ".opc" (and that appears to contain content generated from the OPC Processing library) will be added to the play list, which is sorted alphabetically. Each file is played in turn...at the end of the list, it then returns to the first file. As written, this is limited to 50 files maximum, but that's easily increased in the code if needed.

Good to Know

The files generated by the Processing library are specific to a given installation (e.g. whatever pixel layout you defined using opc.ledGrid() or other methods). Because the OPC library can place pixels *anywhere,* it doesn't convey any data regarding things like matrix size, because it's not inherently tied to matrices (nor any other specific topology).

So, for example, if you render an animation for a 16x16 DotStar matrix with a zig-zag order, it will not play back correctly on an 8x8 matrix, or 32x8, or a progressive pixel order or anything else. *That file will work only for the installation for which it was designed, or one with an identical topology.*