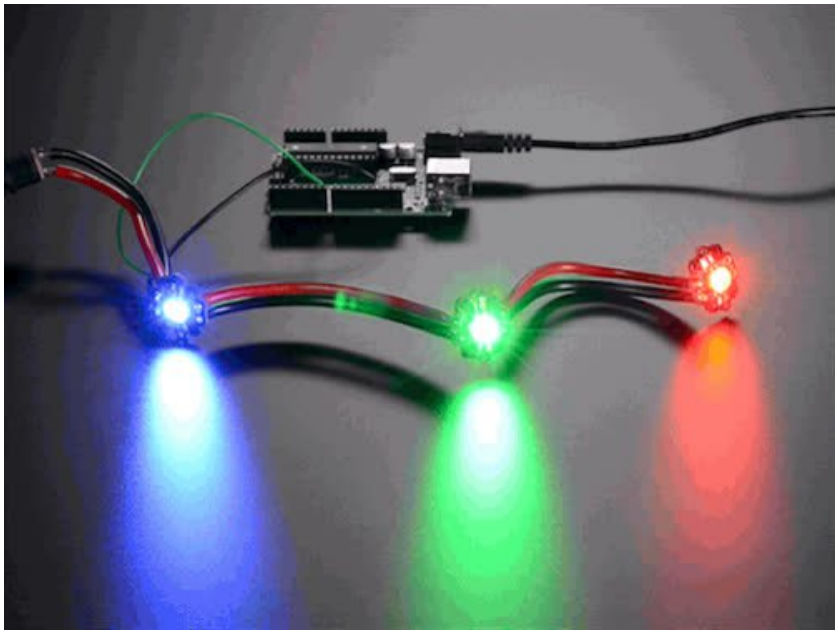
adafruit learning system

Pixie - the 3W Chainable LED Pixel

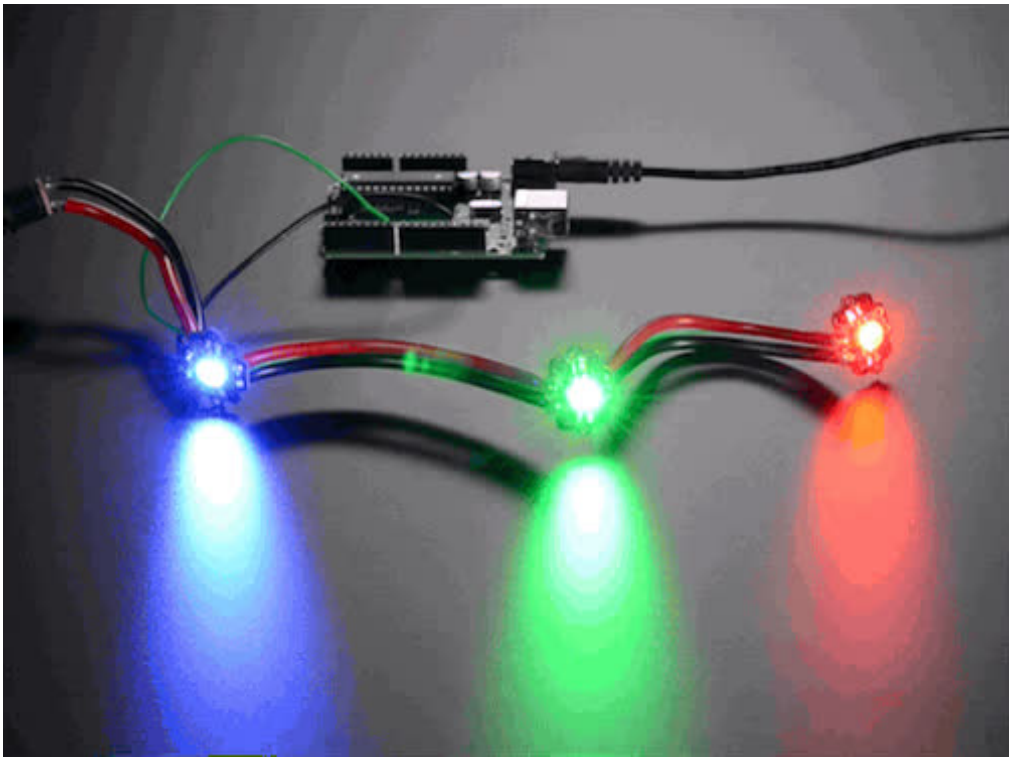
Created by lady ada



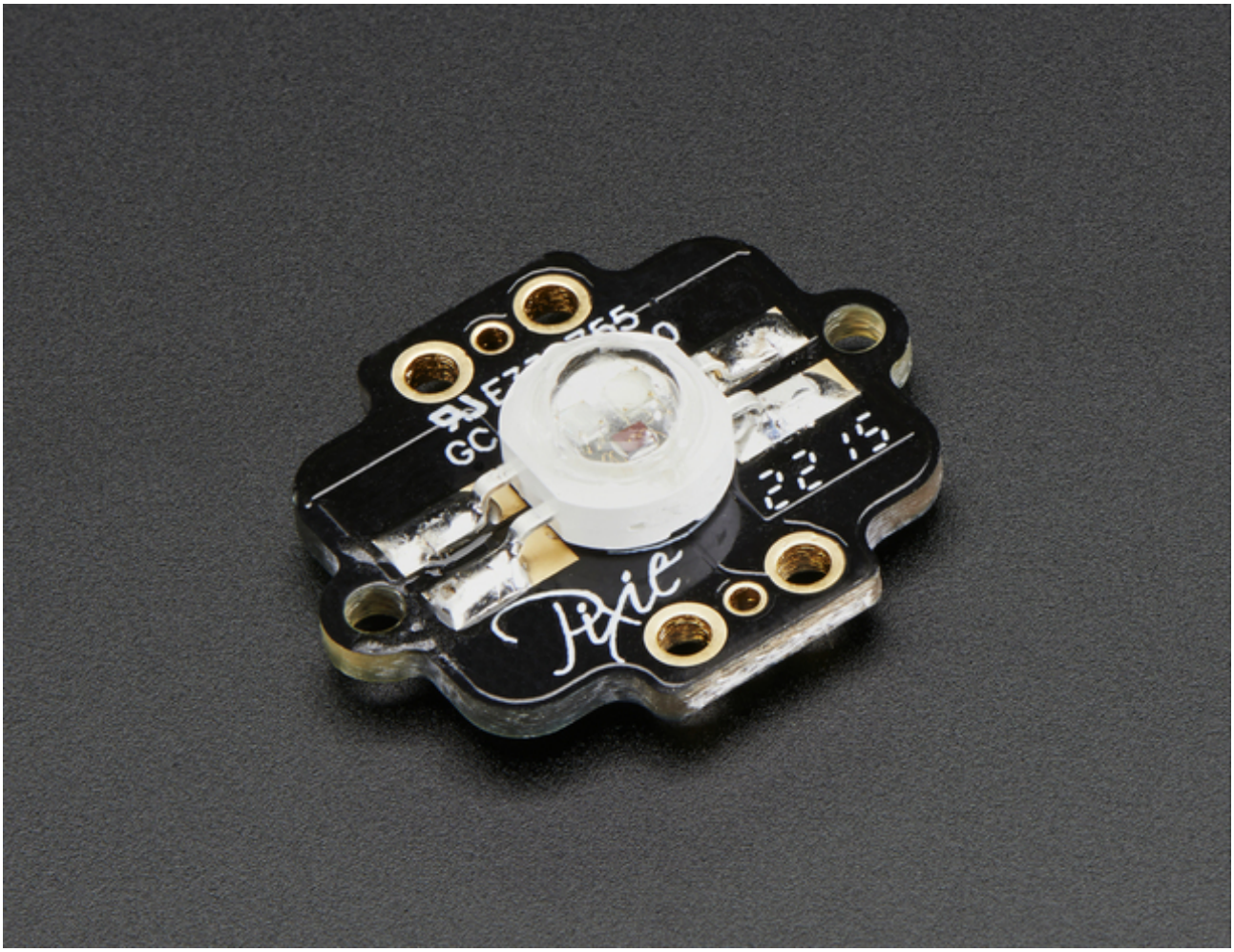
Last updated on 2016-06-03 05:09:19 PM EDT

Guide Contents

Overview

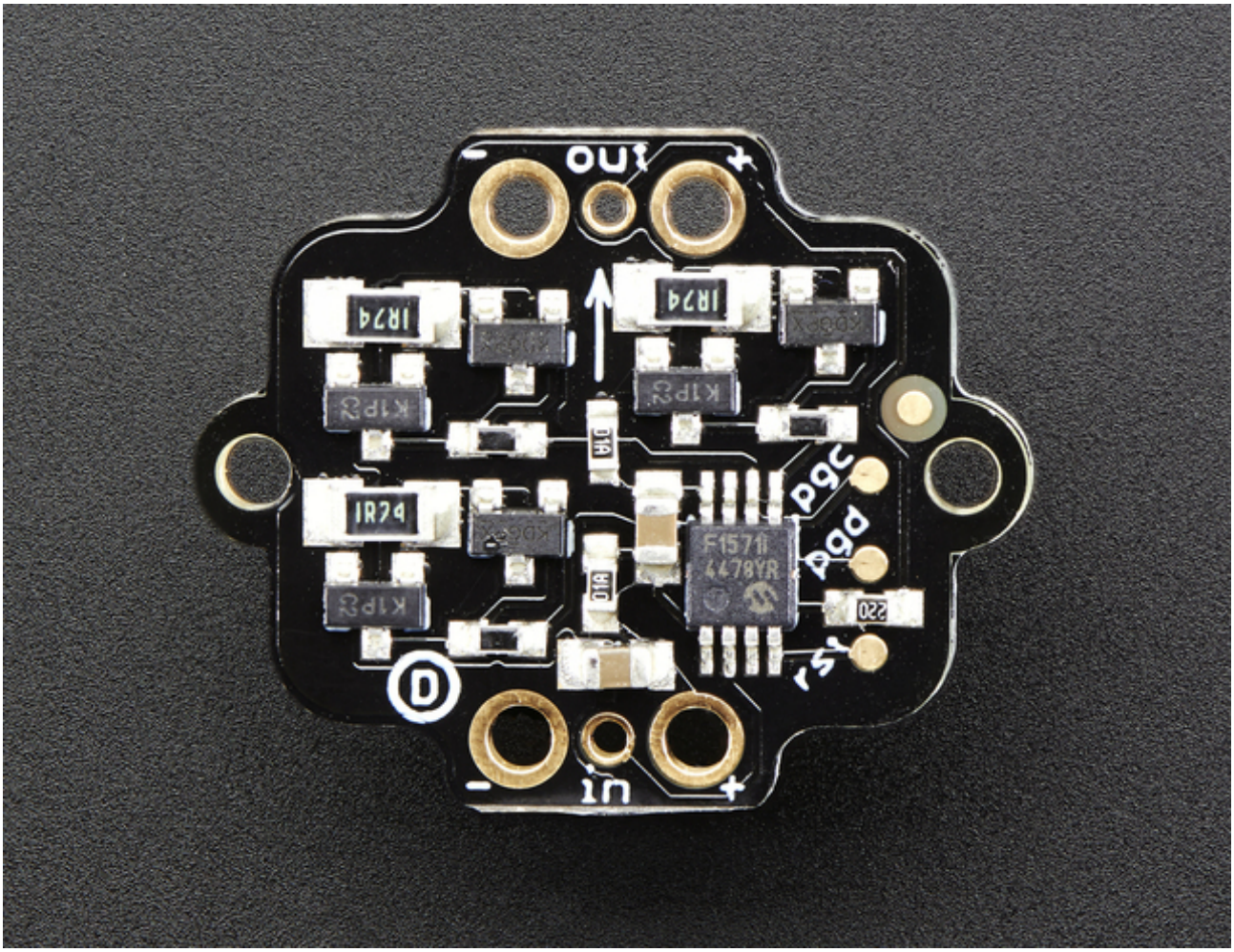


NeoPixels are plenty bright, *suuuure*. BUT ARE THEY 3 WATTS BRIGHT? No! They are not! That's why you need a Pixie. These chainable smart LEDs are not only super-smart, they are ridonkulously bright with 3W total, compared to 0.2W of a 'standard' NeoPixel. Designed by Ytai Ben-Tsvi, these are the ultimate in LEDs.

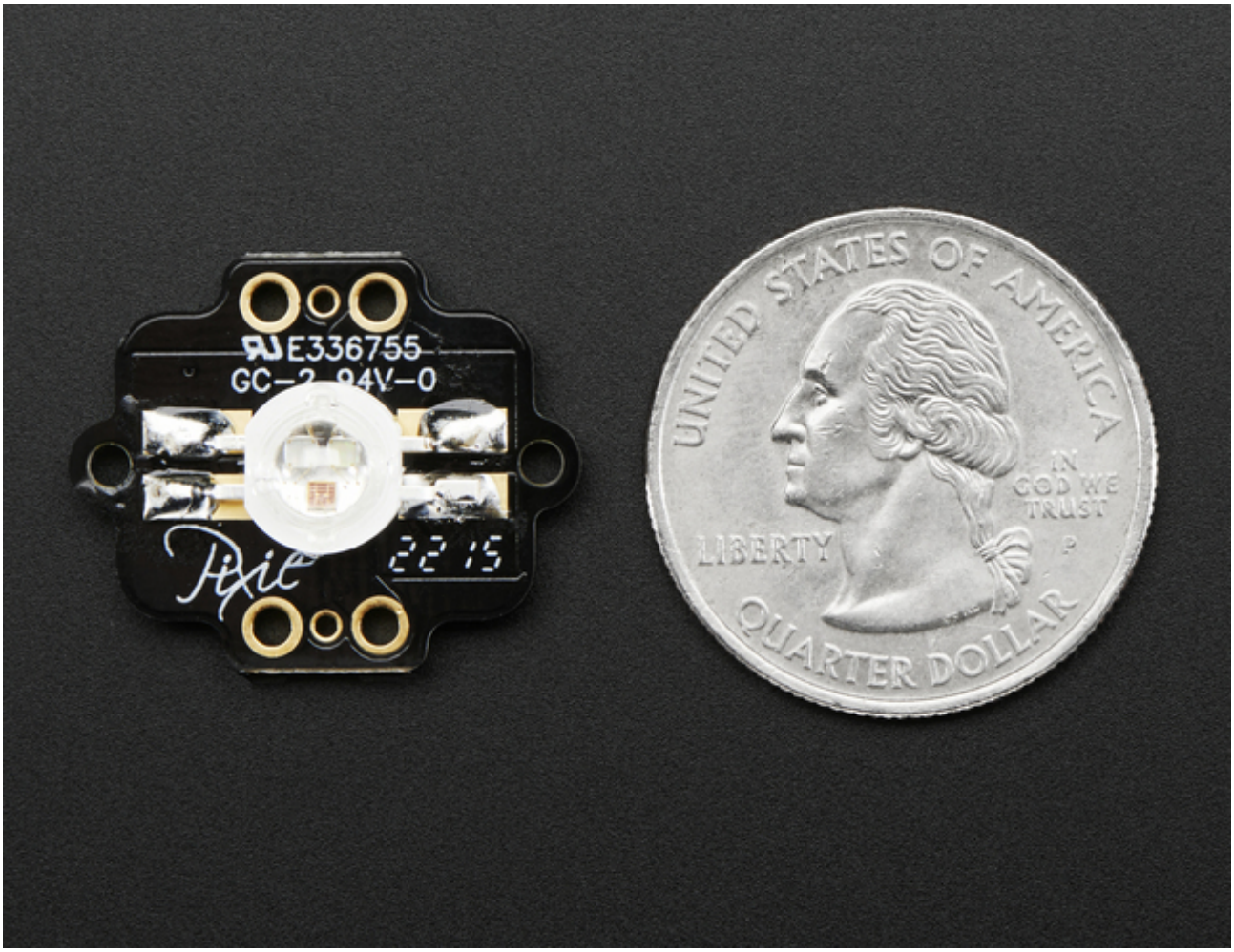


Each Pixie not only contains that aforementioned 3W LED, it also has a Microchip PIC microcontroller. You send it the color you want to appear at standard 115200 baud (1 byte per color). You can send a longer string of pixel data and it will 'forward' along the messages so you can chain them like a shift register. You only have to send the data once per color change. Once set, the microcontroller does all the PWM handling for you.

You do have to send it data every 1 second at a minimum (as a protection against the bright and hot LEDs staying 'stuck on', they will eventually timeout if no updates are received)



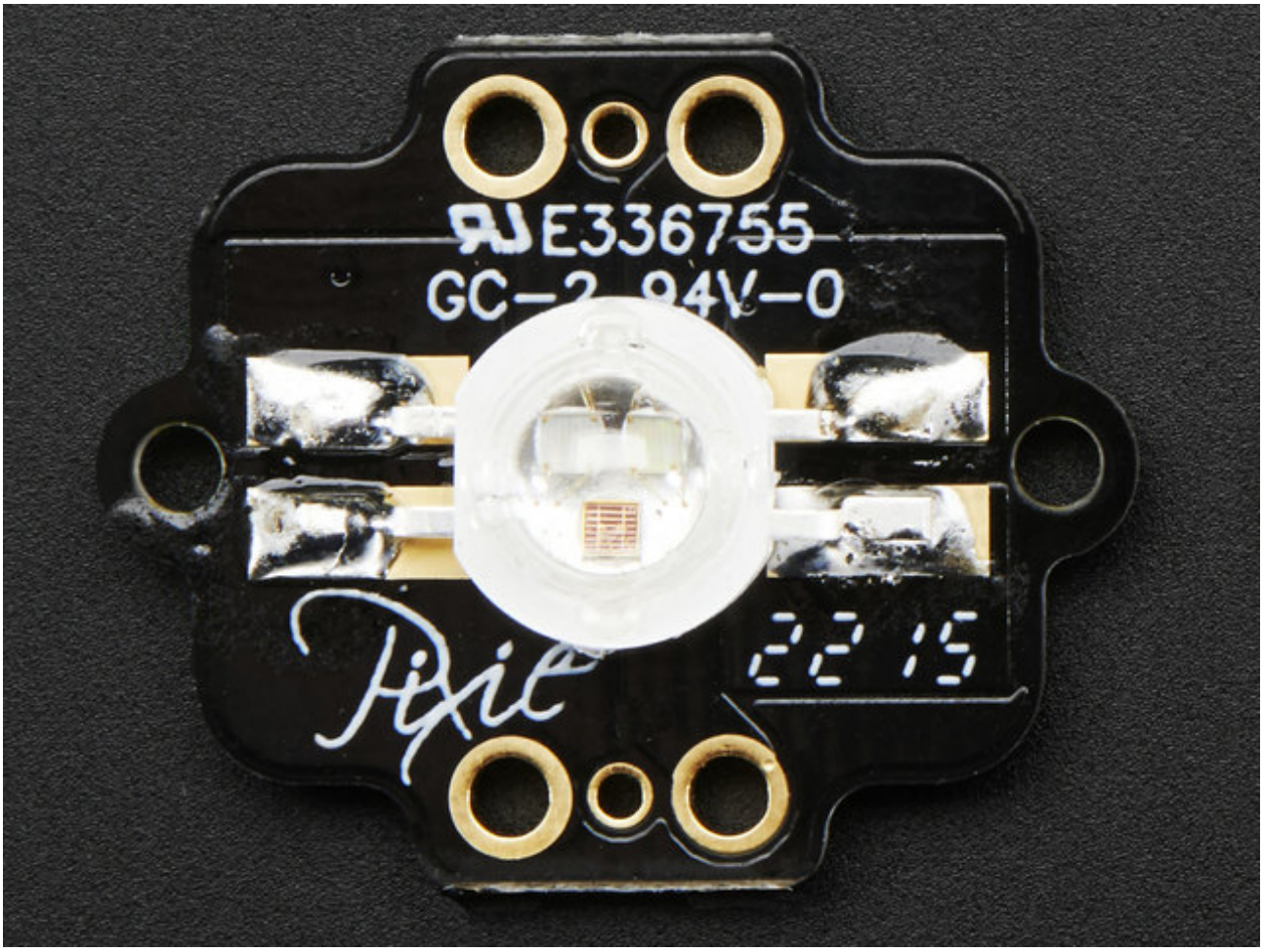
Since the Pixie uses so much power, there's a risk of the LED heating up so much it damages itself and/or the microcontroller. That's why Ytai added a temperature cut-off. When the PIC detects that the Pixie's temperature is too high, it turns off the LED until it gets back down to a reasonable level.

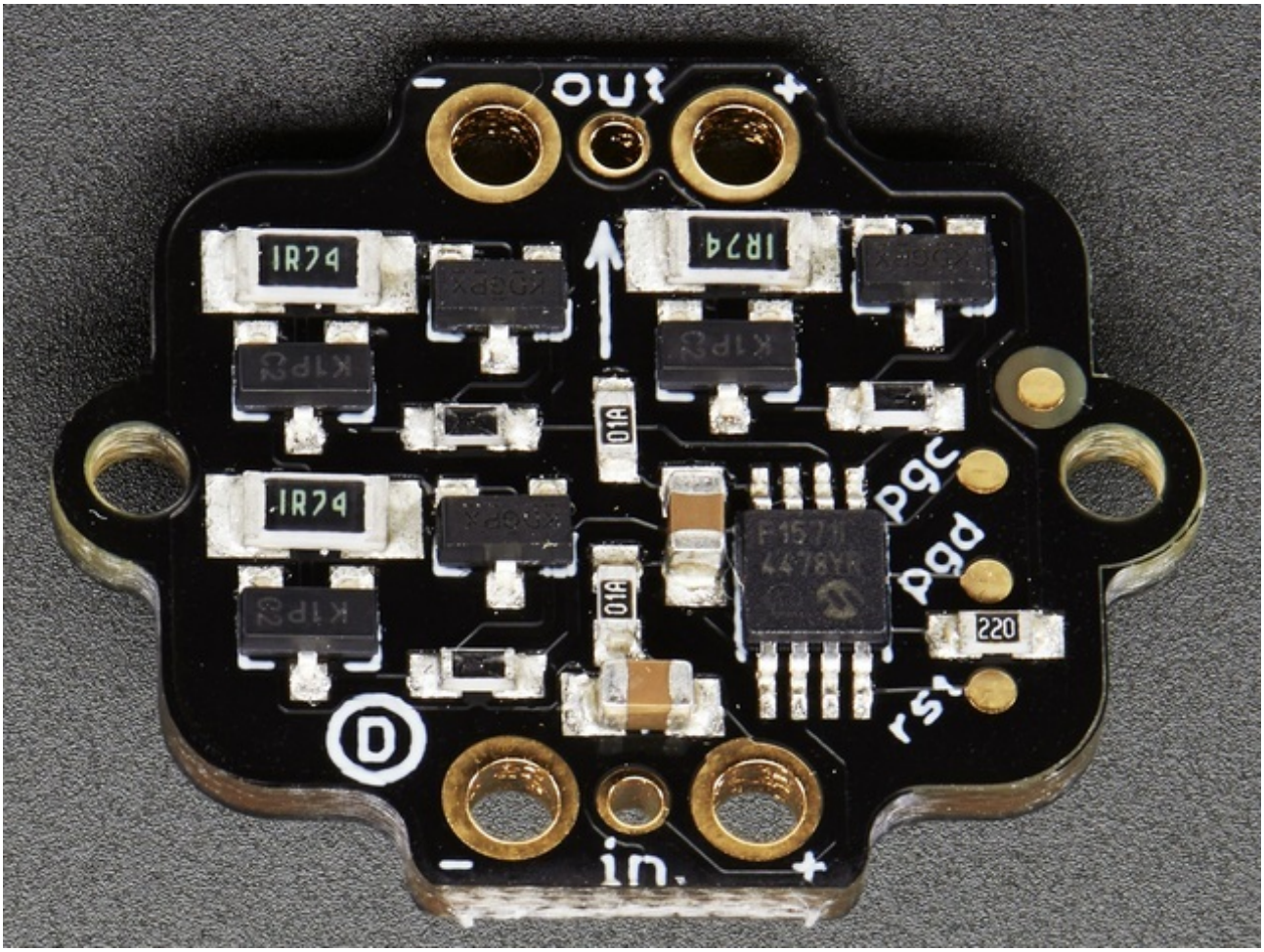


Each Pixie comes fully assembled, with big pads for power wires (don't forget, we're talking about max 1 Amp per Pixie!) and mounting holes on the sides. You'll also need thick wires to power each Pixie and a microcontroller that can send 115,200 baud 8N1 data to the Pixie (on its own, it will not do anything, you must send it data for it to light up)

Our Arduino library is easy to use, has the same API you're used to with our NeoPixel library, and will get you blasting ROYGBIV in 5 minutes or less.

Pinouts





Pixie has two 'ports' - one for input and one for output. Both input and output ports have power pins:

- **+ Power** - This is the positive power and logic pin for the Pixie. Power with 5VDC.
- **- Ground** - This is the ground reference for power and logic pin for the Pixie. Connect to power ground on your power supply as well as the ground of your microcontroller.

Both power pads have a 0.07" / 1.8mm diameter hole, so you can use very thick wires for connecting (16AWG is recommended). That's good because remember that you can draw up to 1A *per Pixie* when the LED is full white.

You also have an **in** and **out** pin. Each are 5V logic in and out, and receive/transmit data at 115200 baud.

All the pins are aligned to a 0.1" grid, so you can solder standard headers to them and use the Pixie on a breadboard or a perf-board.

The mounting holes on both sides are sized to fit M2 hardware.

Design

The Pixie was designed by Ytai Ben-Tsvi, who joins us below with "how he did it"

Don't forget to read the whole thing for a bonus photo at the end!

Take it away, Ytai!

Overview

While having a fairly simple functionality, the Pixie design has more than meet the eye. In contrast to its low-power relatives, the NeoPixels, switching current to a chain of 3W LEDs creates some interesting engineering challenges, which took a few iterations to get right. Here's what we've learned and how we've approached some of the interesting problems.

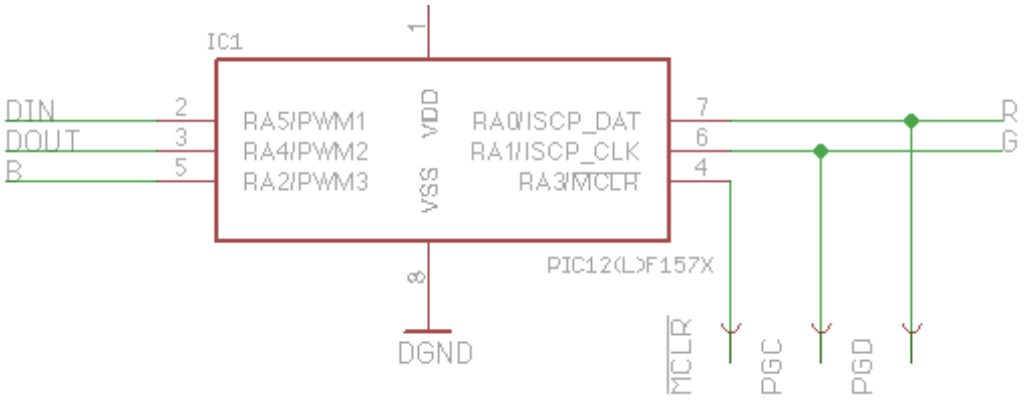
Microcontroller

Fairly early down the design path, it became clear that implementing all the features we wanted is a task most suited for a small microcontroller. We chose the 8-bit Microchip PIC12F1571, which had just about everything we could hope for in this application. It is small and cheap, works on 5V, has exactly 5 I/O pins (used for R, G, B, Din, Dout), an internal oscillator, a 16-bit, 3 channel PWM module, on-die temperature sensor and more. Pretty amazing!

Programming the PIC12 is done through exposed pads featured on the circuit for that purpose (labeled rst/pgd/pgc). A cheap PIC programmer can be used, but the programming protocol is so simple that we've implemented an Arduino library that can do that for our testbed.

The possibilities with having an on-board microcontroller are endless! The Pixie can be reprogrammed for standalone operation, and the Din/Dout pins can be repurposed to support different protocols or to directly connect to buttons, etc. The Dout pin can even be used for analog input!

The exiting firmware can be found in Pixie's Github repository. (http://adafru.it/iNF)

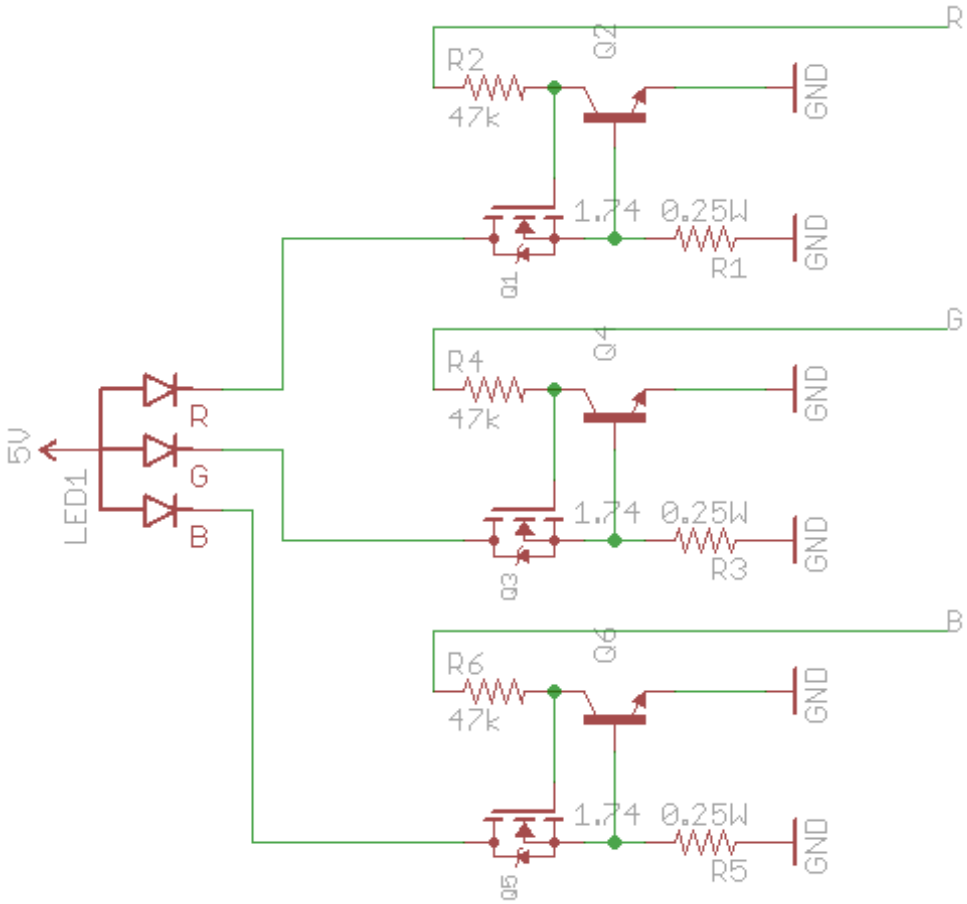


Constant Current Driver

In order to provide a consistent level of illumination each of the R, G, B LEDs needs a constant current supply of about 350mA. We opted for linear regulation for its simplicity and low-cost, despite it being less efficient (and as such, dissipating more heat) than switching regulation.

The constant current circuit is pretty cool. Let's explain it by first considering the path of the current through the LED. The current comes from the 5V supply, through the LED, then through a nFET (Q1/3/5) then through a 1.74[Ohm] shunt resistor. The more resistive the FET becomes between its drain and source, the smaller the current flowing through this path. Now let's see how we can use this to our advantage.

The NPN transistors Q2/4/6 have a specified 0.6V drop between base and emitter when on. This means the voltage across their respective shunt resistors R1/3/5 will always be 0.6V. According to Ohm's law, this means that the current through them will be 0.6[V]/1.74[Ohm], or about 344mA. Close enough. If the current were to decrease, the base voltage would decrease proportionally, resulting in the NPN having more resistance between its collector and emitter, thus causing a higher voltage on the collector (recognize the voltage divider formed between the NPNs and their R2/4/6 pull-ups?). But this would result in a higher voltage on the FET gate, causing it to become less resistive between source and drain and as a result, higher current through the LED. The same logic can be applied in the opposite direction. The conclusion is that this circuit is self-regulating the LED current.



Color and Brightness Control

Different colors are achieved via Pulse Width Modulation (PWM) on each of the R, G, B LED. The PIC has a built-in 3-channel, 16-bit PWM peripheral. This allows us to be fancy and do Gamma correction, which means we are doing a non-linear mapping of the 8-bit color value we are commanded with to a high resolution 16-bit color, resulting in a much more natural color gradient compared to a straight linear mapping. The PWM peripheral runs at about 500Hz. The generated signals switch the constant-current circuit described above.

Daisy-Chaining

Originally, we have designed the Pixie to support the same serial protocol as the WS28x family (aka NeoPixel). It worked. However, this compatibility, which was originally considered a feature has been eventually deemed a drawback: the WS28x protocol doesn't easily lend itself to common micro controller peripherals, and in most cases ends up being bit-banged by the controller, requiring a relative high CPU usage, making it hard to do other timing-sensitive operations at the same time, not to mention driving another chain on a different pin… Our solution: stick to the good ol' 115k.2 asynchronous serial. Almost every microcontroller has a UART peripheral capable of easily generating this protocol without much CPU intervention. Many have more than one. Even a PC with a simple USB-serial dongle can do that fairly easily. Seems like a win! The only drawback we could see what with the data rate being relatively low, we run into frame-rate / chain length limitations (about 60-long chain @ 50 frames/sec). However, at about 1[A] per Pixie, we concluded that typical chains would not be super-long.

The resulting serial protocol is very simple: the controllers sends a byte-string containing a color value for each LED in the chain as follows:

`<R1>, <G1>, <B1>, <R2>, <G2>, <B2>, <R3>, <G3>, <B3>, …, <Rn>, <Gn>, <Bn>, <at least 1ms of silence>`

Each of `<Xi>` is a byte representing the brightness of a single color of a single Pixie, where 0 is off, 255 is fully on, and everything is between is, er, everything in between. `<R1>, <G1>, <B1>` will determine the color of the Pixie that is the first in the chain, counting from the controller end. `<R2>, <G2>, <B2>` is the next one, etc.

Each Pixie listens on it Din pin for serial data. It will consume the first 3 bytes it sees and store them. It will then echo any subsequent bytes to its Dout pin (with less than a microsecond latency). It will keep doing so until it detects a 1ms-long silence on Din. Then, it will immediately apply (latch) the color values it got and go back to listening for a new color. This yield a very effective mechanism for addressing LEDs individually and making sure they all latch at the same time.

Dealing With Supply Noise

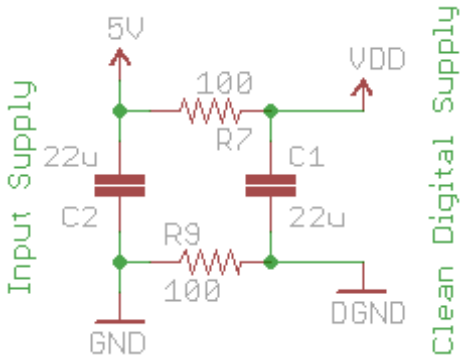
Having a chain with multiple nodes constantly switching 1[A] loads is no small feat! Even an otherwise negligible wire resistance would result in noticeable voltage glitches. Not to mention wire inductance, which likes sudden current changes even less, and reacts with furious voltage surges unless dealt with. To make things worse, being a chain-oriented product, we're expecting people to use rather long

(several meters) wires, which inevitably have more resistance and inductance. And worse still, the on-chip temperature indicator that we really really wanted to use is extremely sensitive to the slightest of noise on the supply. Did we get your attention?

We took several measure to mitigate those issues. First, we made sure the holes for the supply wires are large enough to fit a 16AWG wire. Thicker wires = less resistive = good. The PCB traces connecting the input and output supply are super wide for the same reason.

Then, bulk capacitance! A large 22uF ceramic (hence, low ESR) capacitor across the supply on every node is used to absorb voltage transients, especially those caused by wire inductance. Then, that supply gets further filtered using an R/C circuit comprising R7, R9 and C1, the latter being yet another 22uF ceramic and the relatively high resistor values ensure that the C1 reacts very slowly to any change in the supply voltage. One thing to notice is that we've used to resistors and we let the capacitor "float" in the middle. Why? Assuming fairly equal wires for 5V and GND, this setup would result in the supply rails for all microcontrollers in the chain to always have the same Vcc/2 potential, even if their Vcc voltage is different as result of wire resistance x high current. This makes it easier to discern the 0's from the 1's between consecutive nodes and thus get a reliable communication channel despite the supply noise. Otherwise, since the detection threshold is relative to the supply rails, we would have smaller error margins.

That simple circuit took a lot of tweaking to get right, but the result is very satisfactory noise-immunity characteristics.



Power Dissipation and Over-Temperature Protection

Despite LEDs being relatively efficient light sources, they still convert the vast majority of their consumed power into heat. Furthermore, our linear constant current circuit uses resistance (across the FET) to limit the current, resulting in the extra power being converted to heat. In total, at full steam (driving all 3 LEDs at 100% duty cycle) our little circuit dissipates around 5W! Keeping it from over-heating in this condition is unfeasible. We've allocated largish thermal places on the PCB to improve cooling efficiency, but really, the intention is to not leave the LED full-on for more than a couple of seconds. Rather, working continuously at lower brightness is perfectly fine as well as generating fast, bright pulses periodically.

But we wanted to make sure that the LEDs would not get dangerously hot even by accident. For that reason, the Pixie firmware uses the PIC's on-chip temperature indicator to estimate the board's temperature and would shut-down the LED when it gets too hot (above about 70 degrees celsius). It will automatically resume operation when it cools down. Getting this temperature indicator to work with reasonable precision was a challenge. First, the PIC's supply voltage needed to be extra-clean (as described above) and second, to account for variability between different instances of the PIC, each and every unit goes through an automated calibration process during manufacturing and the temperature calibration data gets written to the PIC's flash memory.

Loss of Communication

Have you ever noticed how NeoPixels retain their color if their controller goes away? While this can be considered a convenient feature in some cases, it is an absolute no-go in a 3W LED case. Losing communications with the controller for any reason during a high-brightness pulse, that was otherwise intended to be very short, could potentially result in LEDs being left on for extended periods, consuming a lot of power and dissipating a lot of heat (limited by the over-temperature feature described above). Even more, what if we have a firmware bug (not that we ever have bugs, but just for the sake of the discussion ;D) causing the PIC to hang while its LED is on? That would be unacceptable.

Watchdog to the rescue! Remember we told you how awesome the PIC12 is? Another feature that is useful for us is the watchdog. It will reset the PIC if it doesn't hear from our firmware that everything is fine for about 2 seconds. In turn, our firmware will only pet the watchdog every time it gets a valid color and successfully latches it. So unless we hear from our controller at least every 2 seconds (and in practice, better leave a little margin), the Pixie resets, causing the LED to turn off until told otherwise.

So unlike NeoPixels, if you want your Pixies to stay on for extended periods, even with no color change, you need to constantly remind them that you're alive by sending them their favorite string.

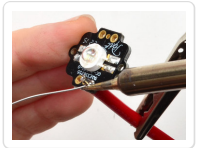
Conclusion

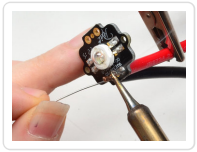
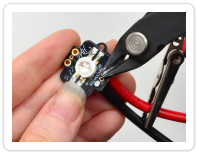
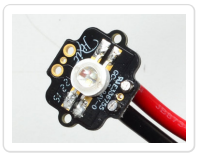
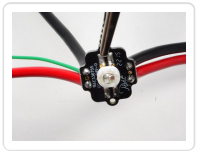
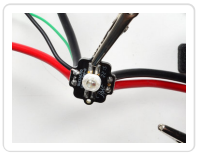
Who would have guessed that designing a circuit having only about 20 simple parts could get so complicated? Certainly not us! We have done our best to provide a high quality, useful product and learned a lot along the way. We're hoping you'll like the result and enjoyed reading about some of the reasoning behind it.

Assembly

Assembling your Pixies isn't too difficult if you have the right tools and materials. Grab you soldering iron, some thin wire (we like silicone covered 26 AWG) and some thick hefty wire, we used 10AWG stranded.

- 
  For the power wires, make sure to get nice hefty ones that can handle the current draw. Of course, it matters how long they're going to be, and how many Pixies are in the chain, and how much they will be lit up. 16 AWG is the recommended size.

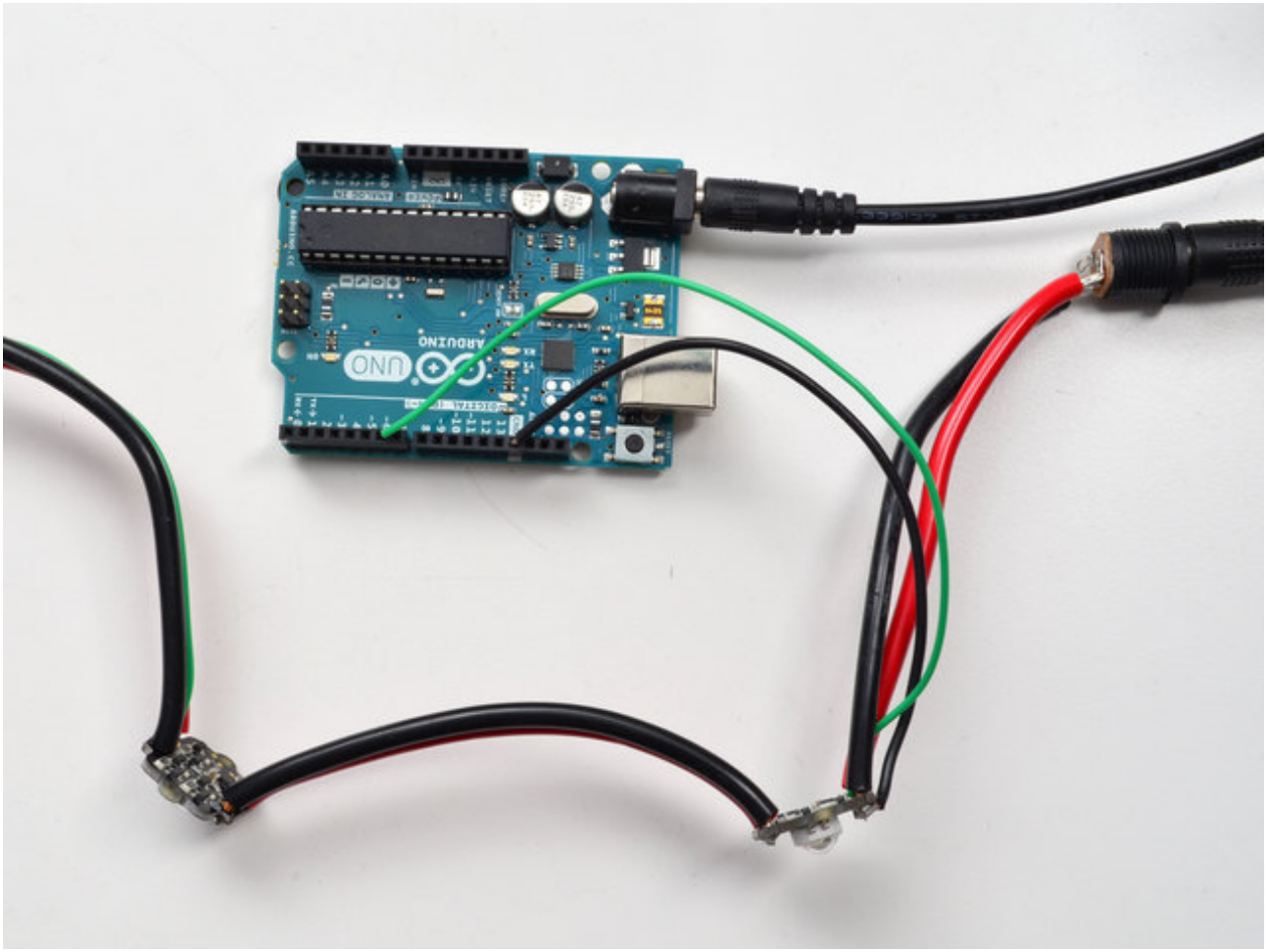
- 
  With thick wires, you may need to get them in place, then tin them and solder into the pad.

-  Don't get + and - mixed up! We suggest red wires for + and black for -
-  You can clip the wires to make them flush, just make sure you dont stress the PCB.
- 
-  You can use a thinner wire for data, in this case we have a ~22AWG data line. It's soldered onto the **out** pin here
-  You may also want a thin wire for your microcontroller ground, especially if you are going to power the Pixies seperately
-  We suggest doing up one Pixie, then testing it. Once you know you're good to go, disconnect it from power and data and solder additional Pixies

  (*please note, in this photo, the green wire is connected to the output! Just check your Pixies before connecting)*

Once you're done with at least on Pixie, go on to the next step where you'll do your test

Wiring & Test



Wiring is easy, you can use any pin with a Serial or Software Serial output pin. We don't use the input of a Serial connection (since the Pixie is 'write only'). Dont forget to power the Pixie with a good 5V supply that can handle the current draw.

Our example code will use digital #6 but you can change this to any pin later

Download Adafruit_Pixie library

To begin reading sensor data, you will need to download Adafruit_Pixie from our github repository (http://adafru.it/iOa). You can do that by visiting the github repo and manually downloading or, easier, just click this button to download the zip

Download the Adafruit_Pixie library

http://adafru.it/iOb

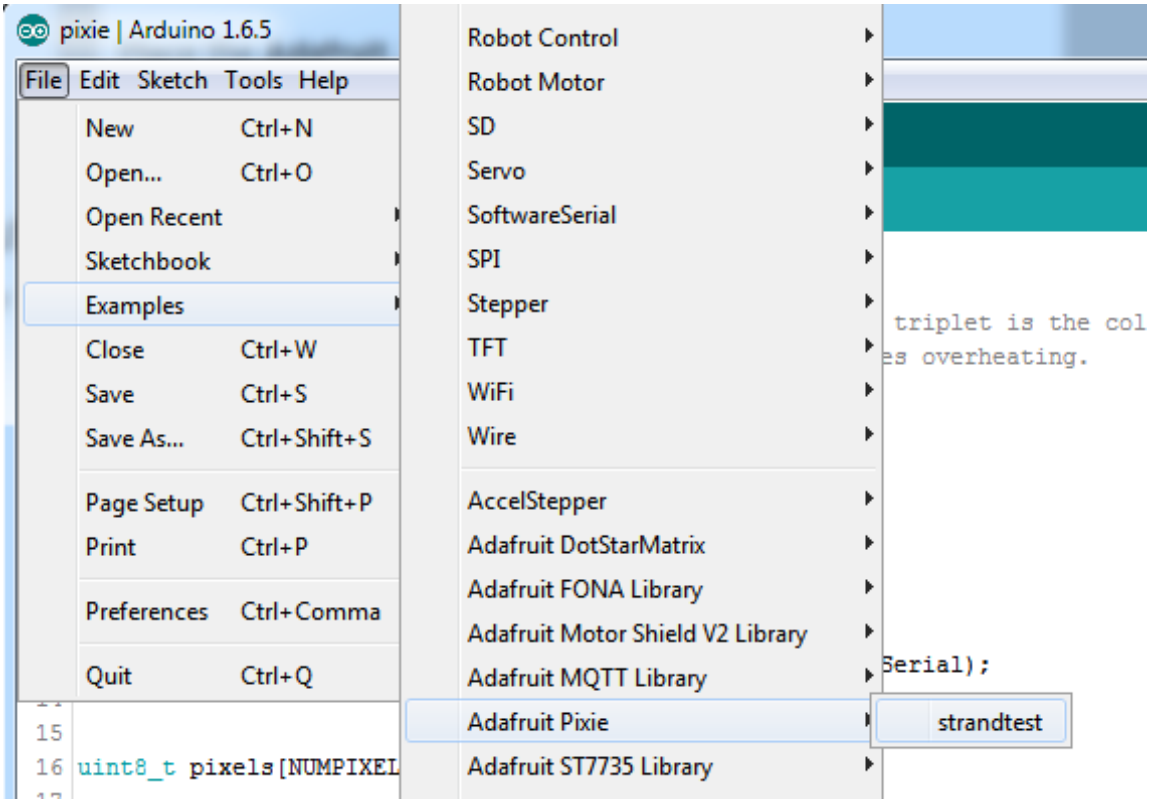
Rename the uncompressed folder **Adafruit_Pixie** and check that the **Adafruit_Pixie** folder contains **Adafruit_Pixie.cpp** and **Adafruit_Pixie.h**

Place the **Adafruit_Pixie** library folder your **arduinosketchfolder/libraries/** folder.
You may need to create the **libraries** subfolder if its your first library. Restart the IDE.

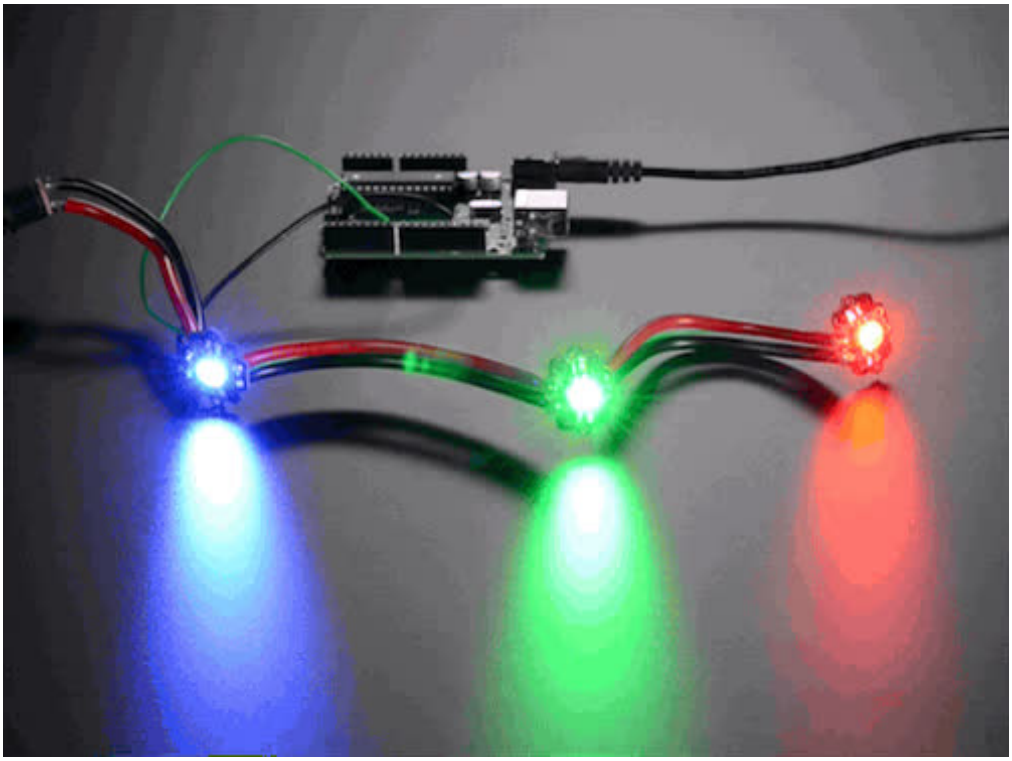
We also have a great tutorial on Arduino library installation at:
http://learn.adafruit.com/adafruit-all-about-arduino-libraries-install-use (http://adafru.it/aYM)

Load Demo

Open up **File->Examples->Adafruit_Pixie->strandtest** and upload to your Arduino wired up to the Pixies



That's it! You're ready to rock out to your bright lights



Library Usage

Start by including both SoftwareSerial and the Pixie Library

```
#include "SoftwareSerial.h"
#include "Adafruit_Pixie.h"
```

Two #define's will determine which pin you're using for controlling the Pixie's and how many there are

```
#define NUMPIXELS 3 // Number of Pixies in the strip
#define PIXIEPIN  6 // Pin number for SoftwareSerial output
```

Pixie's receive data via Serial. You can use hardware serial as well, but our demo is SoftwareSerial. Create a new SoftSerial device that *transmits* on the pixie pin

```
SoftwareSerial pixieSerial(-1, PIXIEPIN);
```

Then pass in the software or hardware serial device to create the Pixie strip

```
Adafruit_Pixie strip = Adafruit_Pixie(NUMPIXELS, &pixieSerial);
```

Start the Pixie object by setting the baud rate, it is 115200 so dont change that number

```
pixieSerial.begin(115200); // Pixie REQUIRES this baud rate
```

You can set the overall brightness, this is a one time 'nonreversable' setting, so once you set pixel colors they will automatically be scaled by the brightness. 0 is all the way off, 255 is all the way on. Default is 255.

```
strip.setBrightness(200); // Adjust as necessary to avoid blinding
```

Then you can set each pixel color with

```
strip.setPixelColor(n, red, green, blue);
```

Where *n* ranges from 0 to numberOfPixels-1 and red, green and blue are the RGB component color, ranging from 0 (off) to 255

For example, to set pixel #1 to full red:

```
strip.setPixelColor(0, 255, 0, 0);
```

To set pixel #5 to a dim green-blue

```
strip.setPixelColor(4, 0, 30, 30);
```

Once the pixel colors are set, you'll need to tell the pixie strand to update, call:

```
strip.show();
```

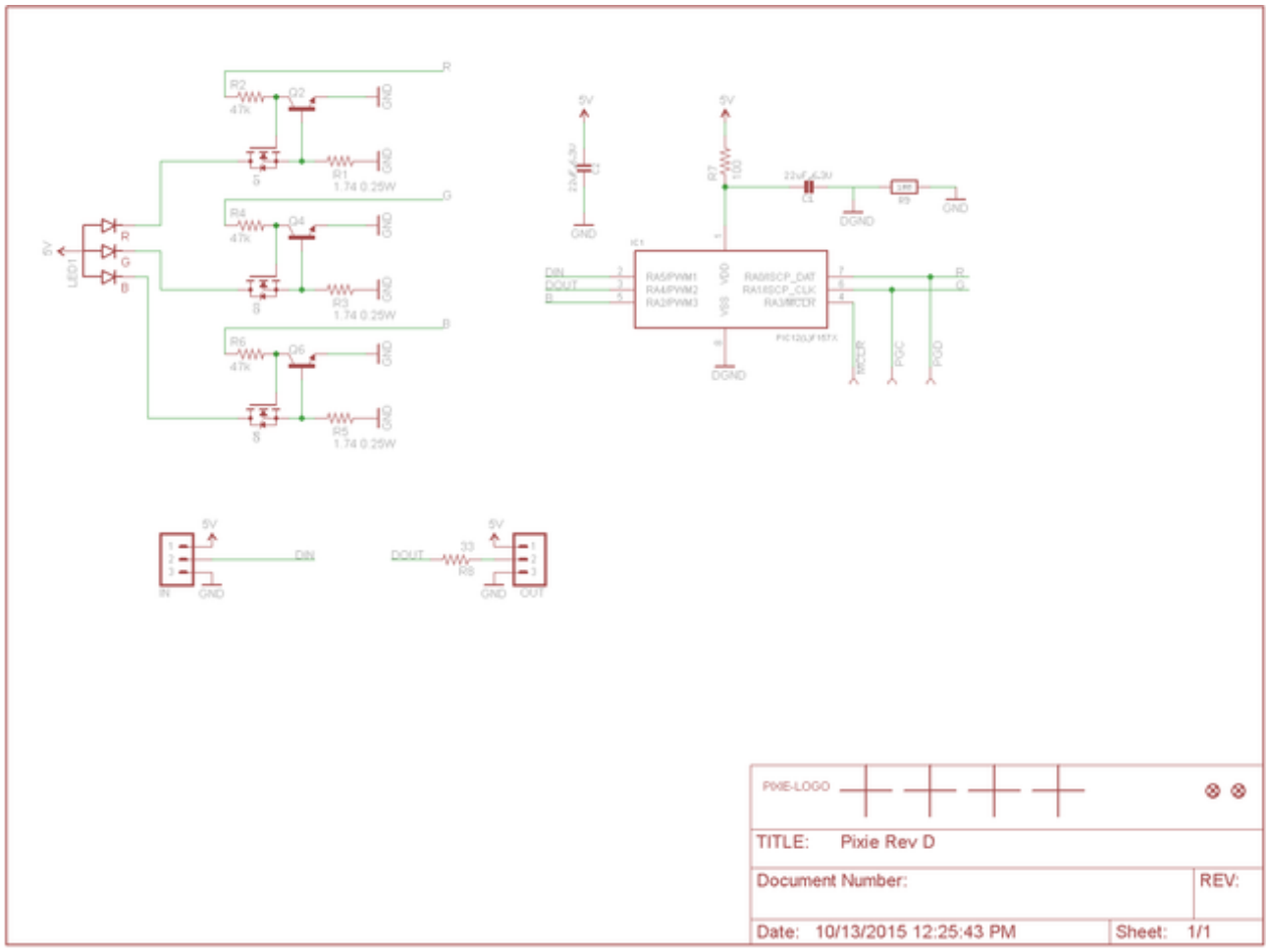
Downloads

Datasheets

- PIC12(L)F157X (http://adafru.it/iEr) - 8-bit PIC microcontrller processor used
- IRLML2060 (http://adafru.it/iEs) - Power transistor for each R G B channel
- 3W RGB LED (http://adafru.it/iEt)

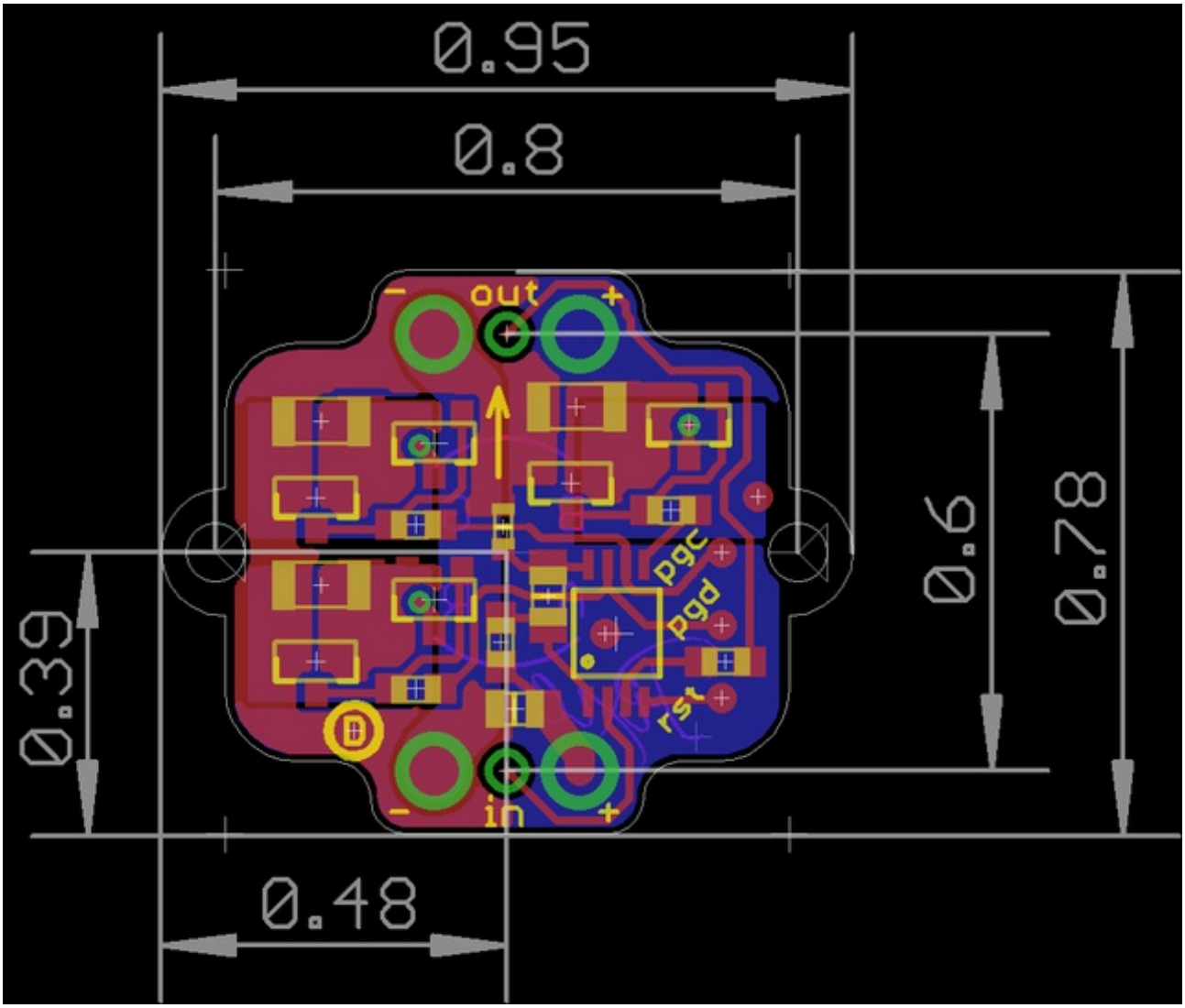
Schematic

Click to embiggen



Fabrication Print

Dims in inches


0.95
0.8
out
pgc
pgd
rst
in
0.6
0.78
0.39
0.48