Build an LED tester

Bruce E. Hall, W8BH

Introduction

I like the challenge of QRP. It is very satisfying to work the world on milliwatts. But I have to admit, it can also be fun to go the other way and crank up the power to a Spinal Tap 11.

The same goes for electronic projects like an LED tester. To check an LED, all you really need is a 3V coin battery. It's simple, cheap, and portable – perfectly matching the cost and complexity of the device under test. The frugal QRP-guy in me approves.

Or, with a Tim Taylor grunt, you can go wild and overengineer it. That's what I did with this LED tester. It's a battery-operated device that shows you current and forward voltage drop for the LED under test. You can quickly choose, without any guesswork, a standard resistor for the current draw and LED brightness you want.

Continue reading if you are interested in building a useful LED tester. I will describe the hardware and software, step-by-step. When finished, you will have a useful piece of equipment for $20 in parts. It may not be as cheap or pragmatic as a coin-cell tester, but it's a lot more fun.

Bill of Materials*

Part	Qty	DigiKey	Amazon	Unit Cost
Seeeduino XIAO	1	1597-102010328-ND	Seeed Studio	$5.40
128x64 OLED Display	1	n/a	HiLetgo, Frienda	$6.40
MCP4261 digipot	1	MCP4261-103E/P		1.58
Rotary Encoder	1	PEC11R-4220F-S0024		1.72
100-ohm resistor, 0.1%	1	YR1B100RCC		0.60
LED (for testing)				

*This table lists parts needed for the steps described below and breadboarding the LED tester. Additional components are required for PCB assembly. See the builder's guide for more information.

Step 1. The Microcontroller

I have been using Arduino hardware for a long time. The original ATmega328 chip has been slowly supplanted by newer and faster devices. For example, I've used Blue Pill and ESP32 microcontrollers in recent projects. The Blue Pill is hard to beat for the money, but works best with a dedicated programmer. The ESP32 is another great choice, but comes in so many flavors that picking the right one can be a chore. For this project I chose a completely different controller: the Seeeduino (that's not a typo) XIAO. It's a chiclet-shaped board, measuring about 18 x 20 mm, that is quite easy to program.

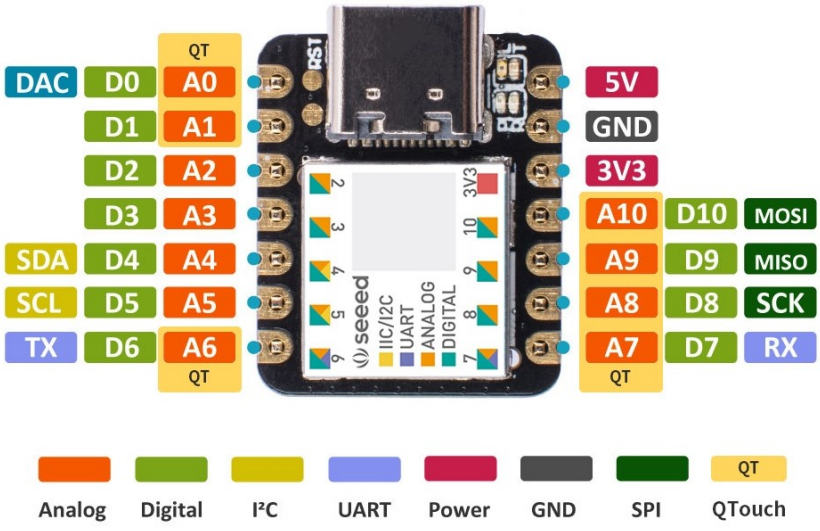
This board is widely available at a cost of $6 each. You can order it on Amazon or directly from Seeed. I ordered mine through Digikey. It is programmed and gets its power through a USB-C cable. So, order one of those, too, if you don't have one already.

The official setup guide is at https://wiki.seeedstudio.com/Seeeduino-XIAO/#hardware. Briefly,

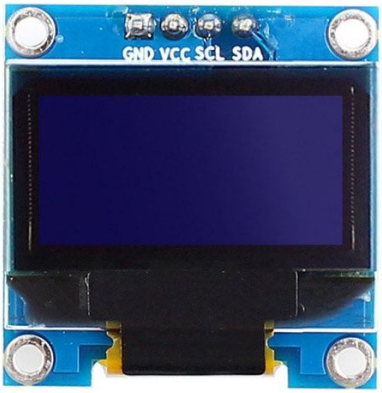
1. Connect the Seeeduino to your computer via a USB-C cable. (My computer does not have a USB-C port, so I use a USB-A to USB-C cable). The green LED on your Seeeduino should light up, indicating that it has power.
2. Start your Aduino IDE application. Go to **File > Preferences** and fill the "Additional Boards Manager URL" with the URL for this board, which is: https://files.seeedstudio.com/arduino/package_seeeduino_boards_index.json
3. Go to **Tools > Board Manager > Boards Manager** and put the words "Seeeduino XIAO" in the search box. You should see an entry for "Seeed SAMD Boards". Click Install.
4. Go to **Tools > Board** and hover over the choice "Seeed SAMD". Select "Seeeduino XIAO".
5. Go to **Tools > Board > Port** and select the com port associated with your device.

Try uploading the blink sketch. The orange LED on the XIAO should flicker as the code is uploaded, then the same LED should blink. During upload the computer will register a USB disconnect followed by a USB reconnect, which is slightly annoying but normal.

For reference, here is the XIAO board pinout:



Step 2. Adding a display



0.96" I2C OLED Display module

For this project, I chose a small, monochrome OLED display. It does not take up much space: the 128x64 pixel display is less than an inch diagonal. OLED technology gives it a crisp-looking output that makes smaller fonts readable. And it is cheap: The HiLetGo version on Amazon is currently $6.40. I got mine on Amazon from "Frienda" at 5 for $16. Nice.

Compared to power-hungry ILI9341 boards, this OLED display with an SSD1306 controller uses very little power. It has an I2C interface, so only 2 interface pins are required. The downside is that your microcontroller must buffer the entire display in its memory. It is not compatible with older microcontrollers that have limited memory. Fortunately, the XIAO has no problems driving this display.

Here are the connections between the XIAO and the OLED display:

For those of us of a certain age, the pin labels on the XAIO sticker are nearly impossible to read. So, refer to the pinout diagram above. Notice that, with the USB jack oriented away from you, there are 7 pins on the left and 7 on the right. The top two pins on either side are unlabeled. Pin D4 is the fifth pin on the left.

Seeeduino XIAO	OLED Display
3v3	VCC
GND	GND
"5" (D5)	SCL
"4" (D4)	SDA

Connect the hardware according to the table above. We will need to write another sketch to test it out. Adafruit has done the hard work and created a display library for our use. Consider returning the favor and buying from them.

Here is a sketch to test your display, available on Github as Step 2a:

```
#include <Wire.h>                       // built-in I2C library
#include <Adafruit_GFX.h>               // Adafruit graphics library
#include <Adafruit_SSD1306.h>           // Adafruit OLED library

#define SCREEN_WIDTH     128            // OLED display width, in pixels
#define SCREEN_HEIGHT    64             // OLED display height, in pixels
#define SCREEN_ADDRESS   0x3C           // OLED display I2C address, in hex

Adafruit_SSD1306 led(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire);

void setup()   {
  led.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS);
  led.display();
}

void loop() {
}
```

Several libraries are required. First, since this display uses I2C for communication, we need to add the Arduino I2C library, called <Wire.h>. Next, we have to add a library for drawing basic graphics and text objects, called <Adafruit_GFX.h>. Finally, we need a library that allows us to communicate with the SSD1306 controller on the display board, <Adafruit_SSD1306.h>.

The Wire library is pre-installed, but you might need to add the Adafruit libraries if you haven't already. In the IDE, go to **Sketch > Include Library > Manage Libraries**. In the search box type "Adafruit SSD1306". If the corresponding entry does not say "INSTALLED", install it now. Do the same for "Adafruit GFX Library".

After the libraries are installed, take note of the screen address define "0x3C". You must enter the I2C address of your display module. Many of these displays use the address 0x3C. Interestingly, the address printed on my display is wrong, and caused me many hours of frustration. The correct address turned out to be 0x3C. If, after looking at the documentation for your screen, you cannot determine its address, download the I2C scanner sketch from GitHub. It will find all of the I2C devices connected to your microcontroller and send their addresses to the serial monitor. Very handy!

After confirming and entering the address of your I2C module, run the sketch. Do you see anything on screen? If not, check your hardware connections again, try the I2C scanner, and confirm the address.

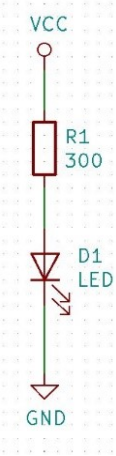
Display text by inserting two lines into the setup() routine:

```
void setup()   {
  led.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS);
  led.setTextColor(WHITE);
  led.print("Hello!");
  led.display();
}
```

Try sketch Step 2b, available on Github, for a more elaborate demo.

It's time to consider LED basics. The next step has no code and no construction. Grab a calculator and get ready for some math instead.

Step 3. Ohm's Law applied to LEDs



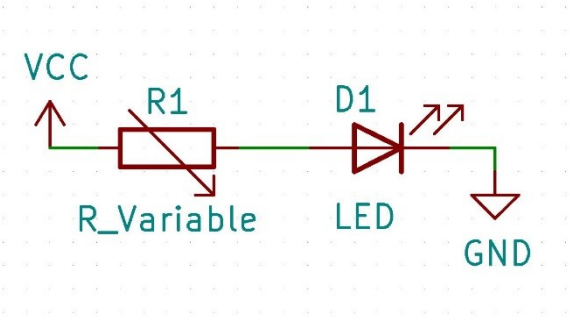
Wikipedia has a good discussion of a simple LED circuit here. You need a voltage source, an LED, and a resistor for limiting current through the LED. The value of the resistor R is determined by three things: the voltage source (Vcc), the forward voltage drop (V_f) across the LED, and the desired current (I) through the LED. The resistance R = (V_{cc} - V_f)/I. LED specification sheets will give you V_f, usually in the 1.8-3.3V range, at a typical current of 20 mA.

In the circuit on the left, let's assume that Vcc = 5.0V and that V_f across the LED is 1.8V. That means that the voltage drop across resistor R1 will be 5.0-1.8 = 3.2 volts. If we use a 300-ohm resistor, the current through the circuit will be 3.2V/300 ohms = 10.7 mA. This circuit will work, at reduced LED brightness, since this current is less than the specified current of 20 mA. If we want maximum brightness at full current, we need a smaller resistor. From the formula above, R = (Vcc- V_f)/I = (5.0-1.8 V)/0.02 A = 160 ohms. Anything smaller will result in too much current and could damage the LED.

Did you follow all of that? Resistor R1 determines the current, and therefore the brightness of the LED. Vary R1 and we vary the brightness. If we substitute R1 with a potentiometer, we have a mechanical way of controlling current/brightness:

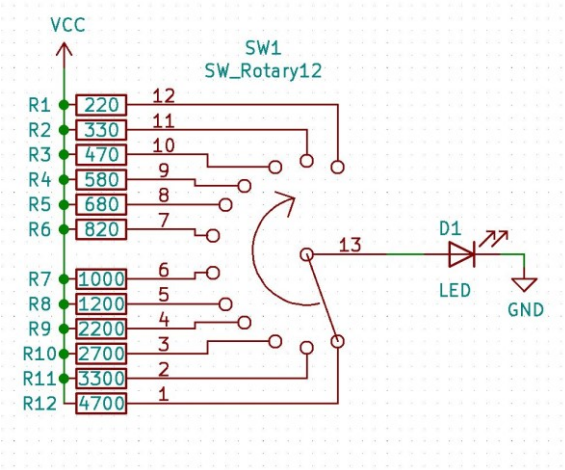
[Side note: variable resistors with 2 terminals are called rheostats. Those with 3 terminals used to divide voltage potential are called potentiometers. We tend to call all 3-terminal devices potentiometers, whether or not they are being used as such.]

This is a good circuit for testing LEDs, as long as you do not crank R1 down too low. For example, a 5K potentiometer in series with a 150-ohm resistor would work nicely for R1.



Suppose we want to know what value of resistor will give the LED a certain brightness (or current draw). We could hook up the circuit above, adjust R1 for the desired brightness, and then measure R1's resistance. It works; try it!

Unfortunately, the measured resistance will probably not conform to a standard resistor value. We can avoid non-standard resistances if we use switched resistors instead of a potentiometer, as shown here.



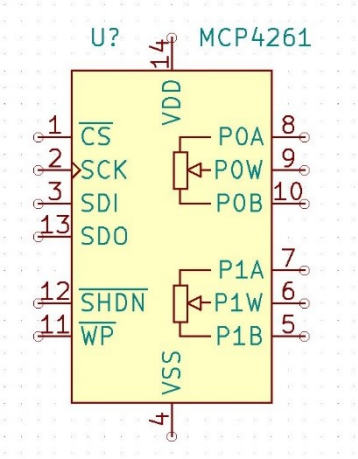
Switched resistors work great, but many more components are required. Fortunately, there is a single IC which can replace all of these added components: the digital potentiometer.

Step 4. The MCP4261 digital potentiometer

A digital potentiometer, or digipot, mimics a variable resistor through the use of a digitally-controlled resistor-ladder network. The MCP4261 digipot provide a variable resistance between 0 and 10K ohms between two of its output pins. The output resistance is quantized into 256 discrete steps. Each step is 10K/256 = 39 ohms. For example, to set the output to 3.9K, you send a value of 100 to the digipot (100 steps * 39 ohm/step = 3.9K). To set the value to 390 ohms, you set the value to 10, etc.

Does it work? Yes, it works very well. But there are a few drawbacks: first, the resistances are specified with 20% tolerance. OK, but not great. Second, the output resistance is the sum of the step resistance, as described above, and the resistance of the wiper terminal. This wiper resistance is nominally 75 ohms, and it is non-linear with voltage and temperature. Without a fixed wiper resistance, it is difficult to accurately compute the potentiometer's output resistance. And finally, this is a low current device, rated at 2.5 mA maximum current through the wiper terminal, lower than typical LED currents.

An ideal wiper resistance would be zero, allowing us to directly compute the desired resistance, without any offset, and without any wiper non-linearity. To reduce the error, I am using a dual-potentiometer chip with both potentiometers in parallel. This reduces the maximum resistance from 10K to 5K, reduces the resistance step size from 39 ohms to 19.5 ohms, and reduces the effective wiper resistance. It also doubles the amount of current that the device can handle.



Here is a diagram for the MCP4261 digipot. This schematic symbol does not show the physical position of the pins; the pins are grouped functionally. The pins on the left are the input pins; the pins on the right are the output pins; and the power pins are top and bottom. Pins 8, 9, and 10 are for potentiometer #0. Pins 5, 6, and 7 are for potentiometer #1. Pins 1, 2, 3, and 13 are for SPI serial communication with the microcontroller.

Time to experiment! Connect your MCP4261 according to the following table. Please note that the XIAO pins D7, D8, and D10 are located on physical pins 8, 9, and 11.

MCP4261 pin	Connects to
1 (CS)	**XIAO pin 8, labeled “7” (D7)**
2 (SCK)	**XIAO pin 9, labeled “8” (D8/SCK)**
3 (SDI)	**XIAO pin 11, labeled “10” (D10/MOSI)**
4 (VSS)	GND
14 (VDD)	+5V
9 (P0W)	Wire to DMM*
10 (P0B)	Wire to DMM*

Connect a digital multimeter to digipot pins 9 and 10 and set it to measure resistance. Turn on the power. What do you see? On power reset, the digipot is set to mid-wiper, which should be about 5K. If you don’t read about 5K, make sure pin 14 is getting 5V and pin 4 is connected to ground.

The digital potentiometer communicates with the microcontroller using its Serial Peripheral Interface (SPI). To use the SPI bus we connect its SPI pins (pins 1-3) to the corresponding SPI pins on the MCU, as shown in the table above. Sending data requires 3 steps in the software:

1. Enabling communication, by takings its ‘chip select’ pin low.
2. Sending data to the device, one byte at a time.
3. Disabling communication, by taking the chip select pin high.

Our MCP4261 uses 2-byte data packets. The first byte is the command byte, and the second byte is data pertaining to the command. Here is corresponding code:

```
#include <SPI.h>                        // use SPI communication with MCP4261
#define DIGIPOT_CS          7           // CS pin on MCP4161 digital potentiometer

void writeDP(byte cmd, byte data) {     // write 2 bytes: 1 command byte & 1 data byte
  digitalWrite(DIGIPOT_CS,LOW);         // enable device write
  SPI.transfer(cmd);                    // send the command byte
  SPI.transfer(data);                   // then send the data byte
  digitalWrite(DIGIPOT_CS,HIGH);        // disable device write
```

```
}
```

We start by including the SPI library and specifying which MCU pin is connected to the digipot (MCU pin D7). Writing a data packet is just 4 lines of code: enable communication, send the command byte, sending the data byte, and disable communication.

We use writeDP() to set the potentiometer's wiper position, and therefore the number of internal resistors in between the two output terminals. For setting potentiometer #0's wiper position, the command byte is 0. So, what resistance would the command writeDP(0,34) create? Using the information above, 34 steps * 39 ohms/step = 1346 ohms.

Using writeDP() above, run the following sketch and measure the resistance.

```
void setup()   {
  SPI.begin();                          // initialize SPI communication
  pinMode(DIGIPOT_CS,OUTPUT);           // use D7 as DigiPot chip select
}


void loop() {
  writeDP(0,0);   delay(5000);          // no resistors = 0 ohms
  writeDP(0,128); delay(5000);          // 128 resistors = 5K
  writeDP(0,255); delay(5000);          // 255 resistors = 10K ohms
}
```

This sketch is available on GitHub as Step 4A. Your DMM should display alternating resistances of approximately 10K, 5K, and 0K ohms. There is a 20% tolerance on these values, meaning that the maximum resistance of 10K could be as low as 8K and as high as 12K. In addition, the wiper itself adds a nominal resistance of 75 ohms but may be as high as 300 ohms, depending on voltage. My DMM gives values of 10.12K, 5.12K, and 120 ohms, respectively. Thus, under these conditions, my wiper resistance is approximately 120 ohms.

Now let's try using both potentiometers in parallel. This gives us half of the maximum resistance, but smaller step sizes and greater current handling. Connect digipot pin-5 (P1B) to digipot pin-10 (P0B), and digipot pin-6 (P1W) to digipot pin-9 (P0W). Now create a new routine, setWiper(), that will set both potentiometers to the same value. The MCP4261 command to set potentiometer #1 is 0x10:

```
void setWiper(byte value) {             // set both digipot wipers to a specified value
  writeDP(0x00,value);                  // set pot#0
  writeDP(0x10,value);                  // set pot#1
}
```

And change loop() to call this setWiper() instead of writeDP():

```
void loop() {
  setWiper(0);   delay(5000);           //  no 19.5-ohm resistors = 0 ohms
  setWiper(128); delay(5000);           // 128 19.5-ohm resistors = 2.5K
  setWiper(255); delay(5000);           // 255 19.5-ohm resistors = 5K
}
```

What resistances do you see now? My DMM shows 5.07K, 2.57K, and 65 ohms. All resistances have roughly halved, including the effective wiper resistance which is now 65 ohms.

We now have all the tools we need to set the resistance of our digipot. To specify any resistance R from 0 to 5K, the wiper must be set to (R – Rwiper)/19.5. Let's create a routine to do that:

```
void setResistance (int r) {            // set pot to a given resistance
   float pos = (r-R_WIPER)/19.53;       // calculate wiper position from R
```

```
   if (pos<0) pos=0;                    // we can't go below zero ohms
   setWiper(round(pos));                // set the wiper position
}
```

Notice how the wiper equation in this routine attempts to correct for wiper resistance. Change loop() again, specifying resistances rather than wiper positions:

```
void loop() {
  setResistance(2000); delay(5000);
  setResistance(5000); delay(5000);
}
```

This sketch is available on GitHub as Step 4B. Run it and check your DMM again. Do you see 2K and 5K resistances? See how close you can get, adjusting the wiper equation according to your specific chip.

Step 5: Display the resistance

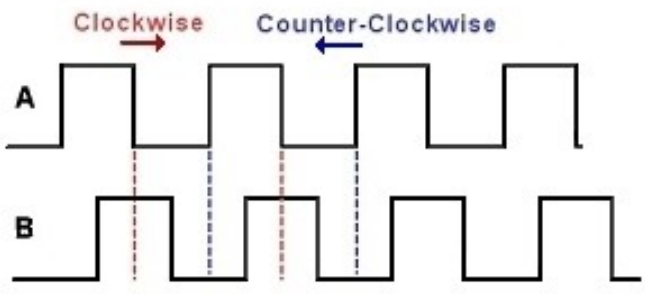
We already have a working display, so let's add the display code back in. Showing the current resistance on the screen is easy. After appropriate initialization of the OLED display, call 'showValue()' to display the current resistance, and call it from within the setResistance() routine:

```
void showValue(int value) {
  led.clearDisplay();                  // erase display contents
  led.setCursor(10,10);                // pick a spot on the display
  led.print(value);                    // write integer value to display
  led.display();                       // and show it
}
```

The full sketch is available on GitHub as Step 5. Run it. Do the OLED display and your DMM agree? Change the resistance values to anything between 100 to 5000 ohms and re-run.

Step 6: Add a rotary encoder

Encoders are the modern-day version of the potentiometer. Indeed, volume controls, which were often potentiometers in the pre-digital era, are often done with encoders today. So how do they work?



The encoder has two outputs, A and B, which are 90-degrees out of phase with

each other. As the knob is turned, each output cycles between high and low. You can determine both motion and direction by comparing the current states with the previous states.

The encoder we are using is an incremental encoder by Bourns. The encoder has 5 pins, 3 on one side and 2 on the other. Considering only the side with 3 pins, connect the center pin to ground and the outside pins to XIAO D0 and D1.

In my Morse Tutor series, I show how to code for a rotary encoder. This time we will take the easy route and use a pre-built library. In the Arduino IDE, go to **Sketch > Install Library > Manage Libraries**. Search for "RotaryEncoder" (no spaces) by Matthias Hertel and install it. Try the following code, available as Step 6a:

```
#include <RotaryEncoder.h>                        // Matthias Hertel version
#define ENCODER_B           0                     // Connect to MCU pin D0
#define ENCODER_A           1                     // Connect to MCU pin D1

RotaryEncoder  encoder(ENCODER_A, ENCODER_B);     // instantiate encoder object

void setup()   {
  Serial.begin(115200);                           // init serial monitor at 115200 baud
}

void loop() {                                     // set alternating resistances:
  static int oldPos=0;                            // previous position
  encoder.tick();                                 // keep encoder states current
  int pos = encoder.getPosition();                // get current encoder position
  if (pos!=oldPos) {                              // has encoder moved?
    Serial.println(pos);                          // if so, show new position
    oldPos = pos;                                 // and remember it
  }
}
```

The four important lines are highlighted. First, include the library. Then, create and initialize the encoder object, giving it the pin numbers for its two data lines. In loop() you must do two things: continuously update the encoder's status with tick(), and then get the encoder's position with getPosition().

Run this sketch and open your serial monitor. As you turn the encoder knob, the "position" of the encoder will be shown as an integer. Turn one way, the number increases. Turn the other way, the number decreases. If you don't see this effect, check your hardware connections.

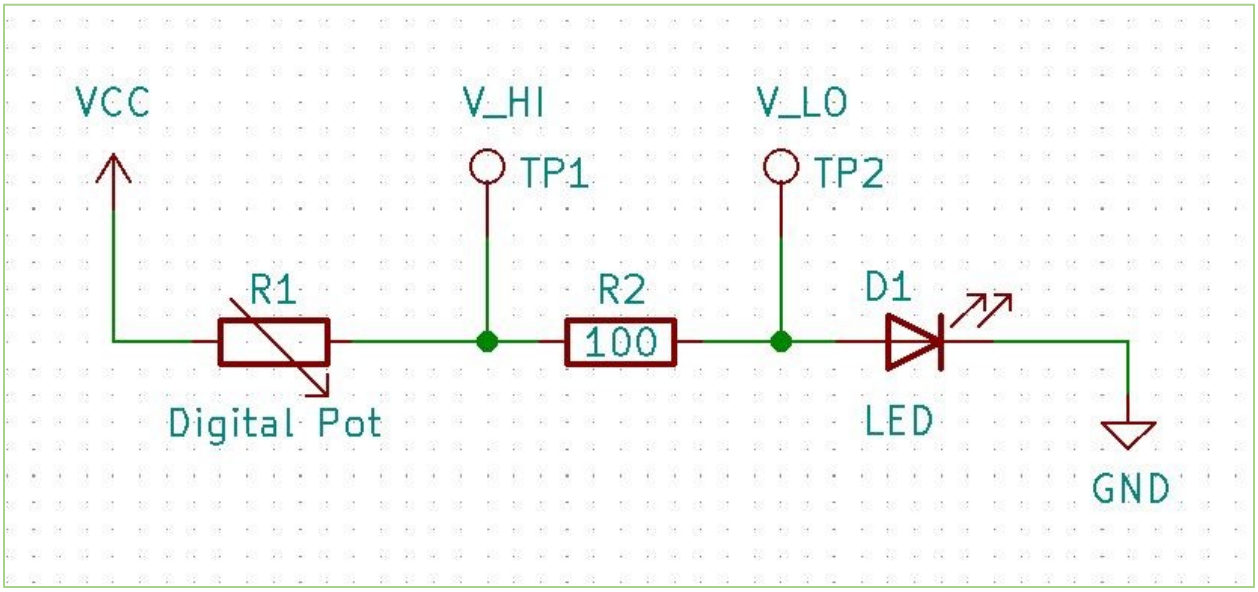
Now we can use our encoder to control the digipot's resistance. Can you combine this code with the step5 sketch? The combined sketch is available on Github as Step 6b. The new loop() becomes:

```
void loop() {
  static int oldPos = 0;                          // previous position
  encoder.tick();                                 // keep encoder states current
  int pos = encoder.getPosition();                // get current encoder position
  if (pos != oldPos) {                            // hsa encoder moved?
    setResistance(pos*100);                       // set resistance in 100-ohm steps
    oldPos = pos;                                 // and remember it
  }
}
```

The only difference is calling setResistance() instead of a Serial.print(). Run it with your DMM connected. Turn the encoder knob, and verify that the resistance displayed on the OLED matches what your DMM sees.

Step 7: Measuring LED voltage and current

Going back to the schematic in Step 3, let's add a fixed resistor R2, and two voltage test points above and below it, called V_{HI} and V_{LO}:



The LED forward voltage, V_f, is the voltage difference across its leads, which is (V_{LO} – GND). Since GND is at zero potential this simplifies to $V_f = V_{LO}$.

How can we determine the current flowing through the resistor, I_{LED}, without our handy DMM? Remember that in series circuit, the same current flows through each component, so $I_{R1} = I_{R2} = I_{LED}$. From Ohm's Law, we know that current flowing through the fixed resistor R2 = V/R. The resistance is 100 ohms, and the voltage across R2 is V_{HI} - V_{LO}, therefore, $I_{LED} = I_{R2} = (V_{HI} – V_{LO})/100$.

Breadboard the circuit above, removing the DMM and applying a Vcc = 3.3V to digipot pin 10. Digipot pin 9 is connected to a 100-ohm resistor, which is connected to the anode of a common red LED, and the LED cathode is connected to ground. Make sure to use 3.3V in this circuit, not 5V. Run the sketch. The LED should light, and should get brighter as you turn the encoder knob to decrease resistance. Now measure the voltage across the LED with your DMM. At a resistance of 2000, I read about 1.77V across the LED. This forward voltage gradually increases to about 1.91V as I decrease the resistance. Most red LEDs have a V_f of 2V at maximum brightness, and slightly less when dim. What do you read?

Measure the voltage V_{HI} and V_{LO} at different resistance values, and create a table like this:

Resistance	V_HI	V_LO	(V_HI – V_LO)/100
2000	1.84	1.77	0.0007 A = 0.7 mA
1000	1.93	1.80	0.0013 A = 1.3 mA
500	2.08	1.83	0.0025 A = 2.5 mA
200	2.36	1.88	0.0048 A = 4.8 mA
100	2.61	1.91	0.0070 A = 7.0 mA

V_{LO} represents the forward voltage across the LED. The last column, calculated from V_{HI} and V_{LO}, represents the current flowing through the LED.

All we need is a way to measure V_{HI} and V_{LO}. Fortunately, the XIAO has built-in analog-to-digital conversion. Connect these test points to ADC input pins on the microcontroller and Bob's your uncle.

Well, almost. To get the value of an analog input we use an Arduino routine called analogRead(). Pass it the MCU pin that you are reading, and it returns a value from 0 to 1023, where 0 = 0 volts and 1023 = 3.3 volts. To convert this value to volts, multiply by 3.3 and divide by 1023.

Let's create a simple routine that returns the voltage on any input pin:

```
float getVoltage(int pin) {                    // returns voltage on an input pin, 3.3V max
  int data = analogRead(pin);                  // get analog input, returns range 0..1023
  return (3.3*data)/1023;                      // convert this value to voltage
}
```

Let's try measuring voltages with our microcontroller. Connect XIAO pin 2 to V_{HI} and pin 3 to V_{LO}. Once per second, we will measure both voltages and show them on the display:

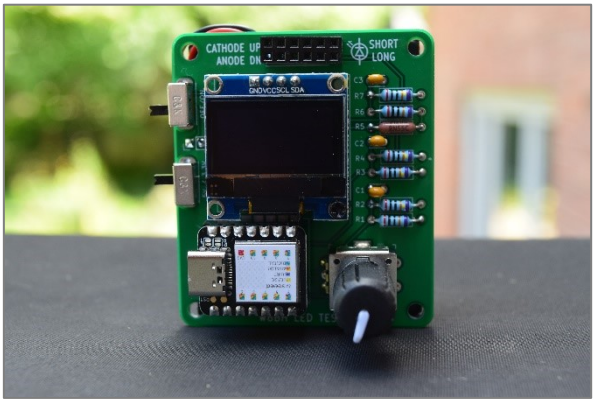
The complete sketch is available on GitHub as Step 7. Run it. Turn the rotary encoder and watch how the values of R, V_{HI}, V_{LO}, and current change. Do the voltages and current closely match what you measured with your DMM?

Finishing touches

We have a working LED tester which shows current and Vf – for any given resistance – with less than 100 lines of code. The final sketch adds a few touches, such as common-value resistors and checking for short-circuits. The PCB adds a few features of its own: battery operation and selectable 3.3V/5V testing. What features will you add?

Reference

1. This document on w8bh.net
2. Builder's Guide
3. Schematic
4. Source code on GitHub
5. PCB Gerbers



Last updated 10/10/2021