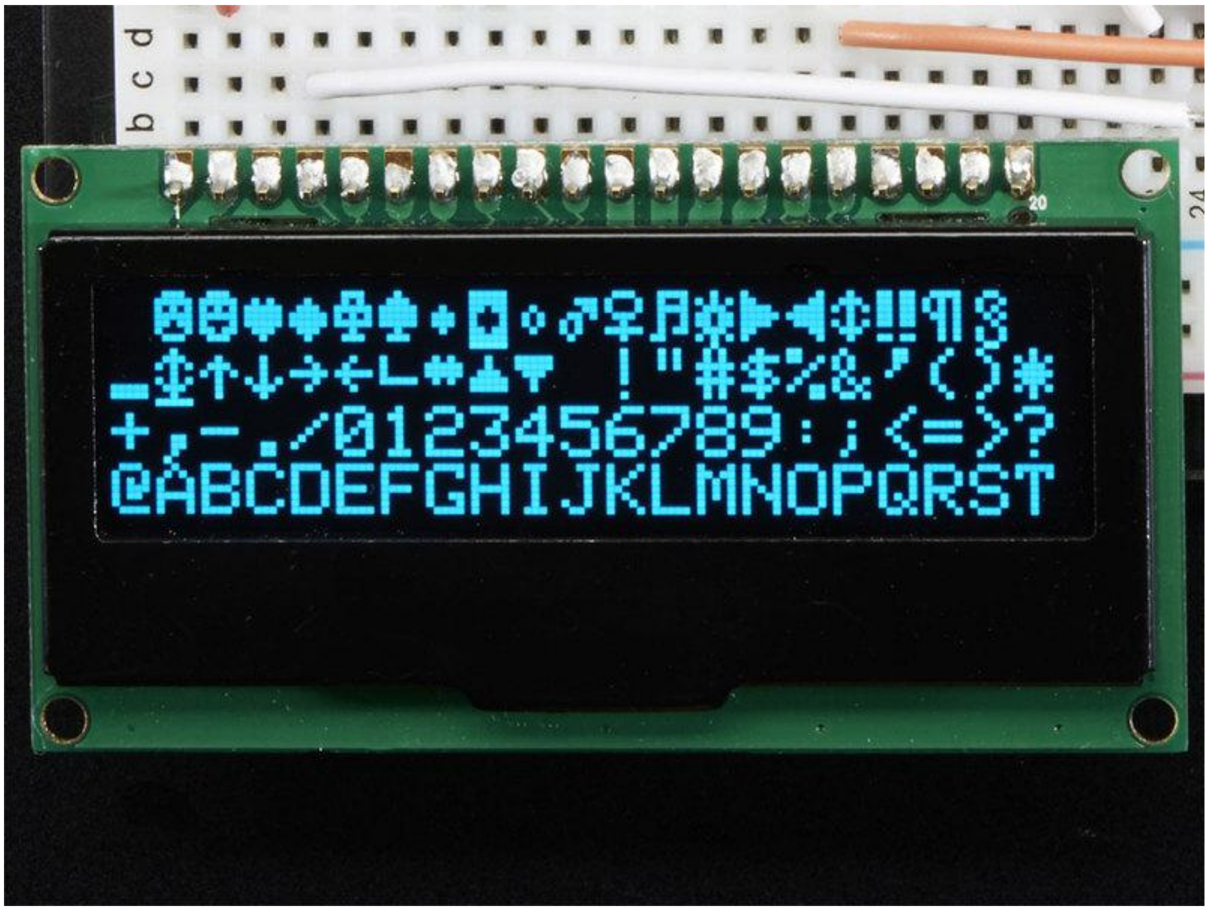
2.3" Monochrome 128x32 OLED Display Module

Created by lady ada

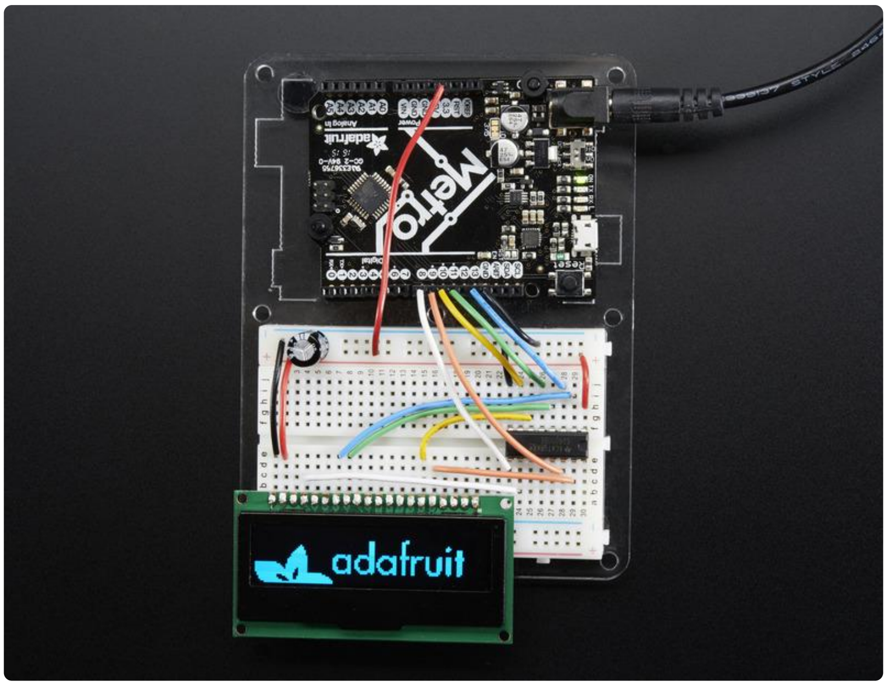


https://learn.adafruit.com/2-3-monochrome-128x32-oled-display-module

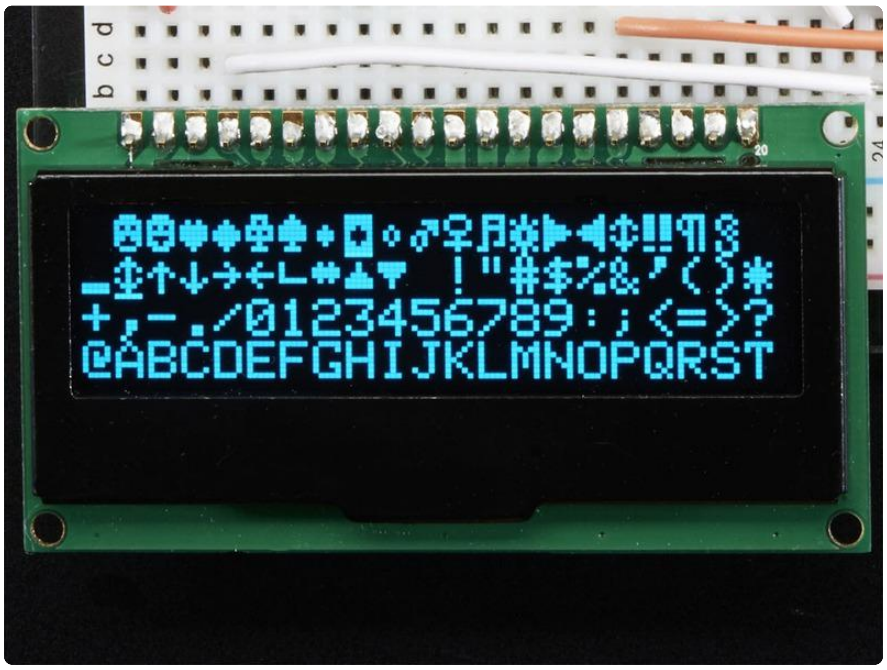
Last updated on 2022-12-01 02:35:43 PM EST

Table of Contents

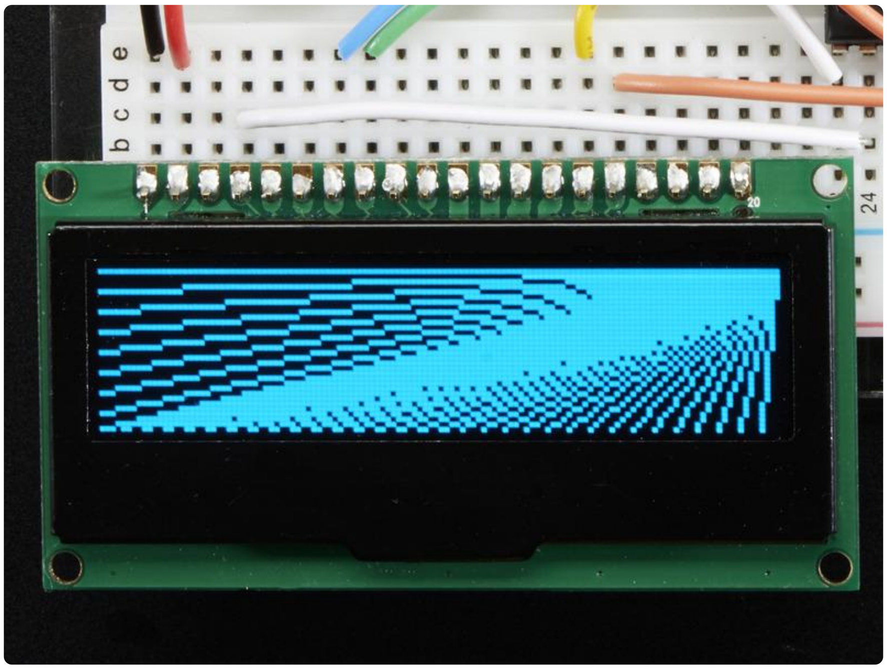
Overview



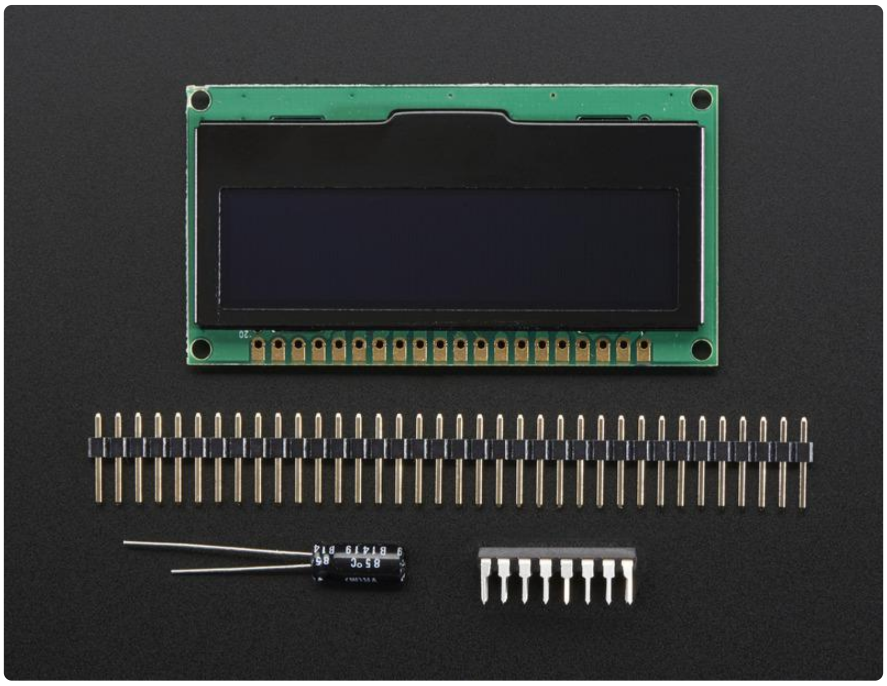
If you've been diggin' our monochome OLEDs but need something bigger, this display will delight you. These displays are 2.3" diagonal, and very readable due to the high contrast of an OLED display. This display is made of 128x32 individual blue OLED pixels, each one is turned on or off by the controller chip. Because the display makes its own light, no backlight is required. This reduces the power required to run the OLED and is why the display has such high contrast; we really like this graphic display for its crispness!



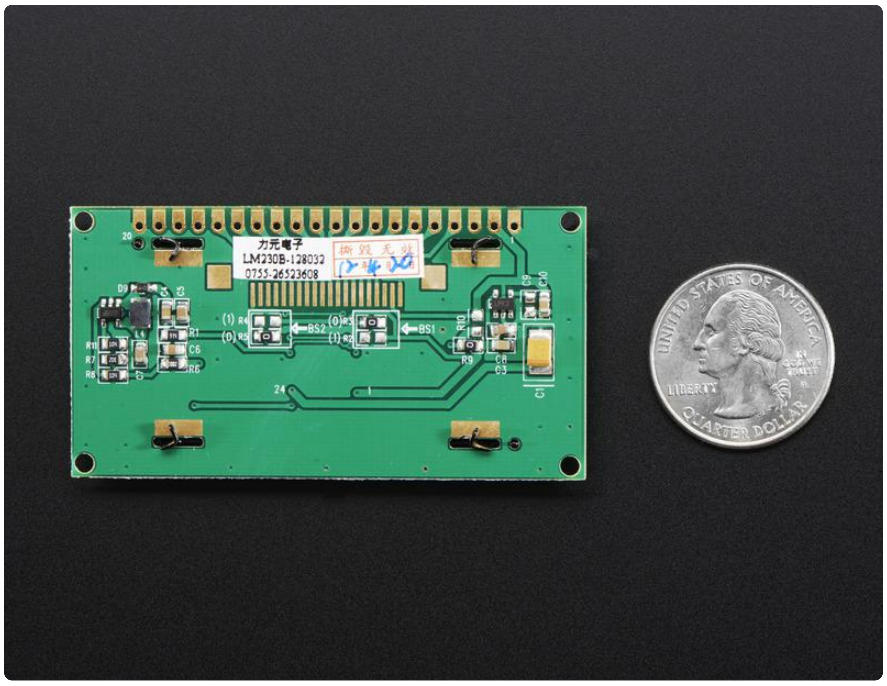
The driver chip, SSD1305 can communicate in three ways: 8-bit, I2C or SPI. Personally we think SPI is the way to go, only 4 or 5 wires are required and its very fast. The OLED itself requires a 3.3V power supply and 3.3V logic levels for communication. We include a breadboard-friendly level shifter that can convert 3V or 5V down to 3V, so it can be used with 5V-logic devices like Arduino.



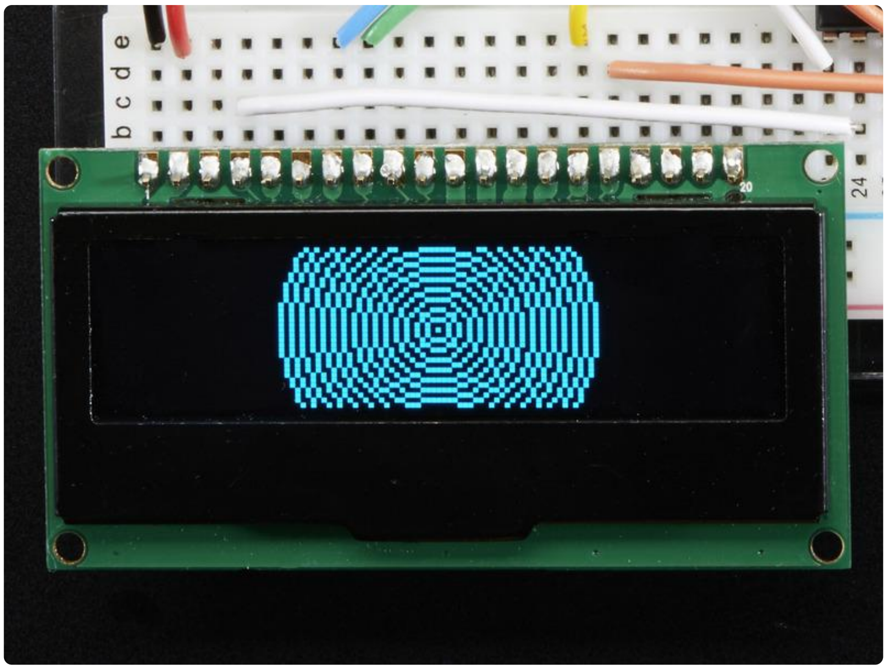
The power requirements depend a little on how much of the display is lit but on average the display uses about 50mA from the 3.3V supply. Built into the OLED driver is a simple switch-cap charge pump that turns 3.3V into a high voltage drive for the OLEDs.



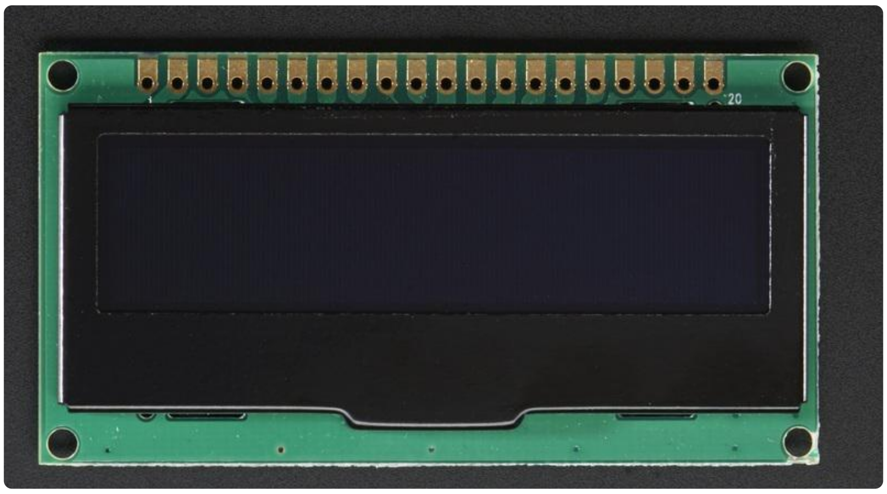
Each order comes with one assembled OLED module with a nice bezel and 4 mounting holes. The display is 3V logic & power so we include a HC4050 level shifter. We also toss in a 220uF capacitor, as we noticed an Arduino may need a little more capacitance on the 3.3V power supply for this big display! This display does not come with header attached but we do toss in a stick of header you can solder on. Also, the display may come in 8-bit mode. You can change modes from 8-bit to SPI with a little soldering, check out the tutorial for how to do so. ()



Getting started is easy! We have a detailed tutorial and example code in the form of an Arduino library for text and graphics. () You'll need a microcontroller with more than 512 bytes of RAM since the display must be buffered. The library can print text, bitmaps, pixels, rectangles, circles and lines. It uses 512 bytes of RAM since it needs to buffer the entire display but its very fast! The code is simple to adapt to any other microcontroller.



Pinouts



The pins on these modules are not well marked, but the one on left is #1 and the pins increment in order until the one on the very right, #20

Power Pins

- Pin #1 is power and signal Ground
- Pin #2 is 3V Power In - provide 3V with 50-75mA current capability
- Pin #3 is not used, do not connect to anything

Signal Pins

- Pin #4 is DC - the data/command pin. This is a 3V logic level input pin and is used for both SPI and 8-bit connections
- Pin #5 is WR - the 8-bit write pin. This is a 3V logic level input pin and is used for 8-bit connections. Do not connect if using SPI/I2C
- Pin #6 is RD - the 8-bit read pin. This is a 3V logic level input pin and is used for 8-bit connections. Do not connect if using SPI/I2C
- Pin #7 is Data0 - this pin is the SPI Clock pin, I2C Clock pin and the 8-bit data bit 0 pin. This is a 3V logic level input pin when used with I2C/SPI, and an input/output when used in 8-bit.
- Pin #8 is Data1 - this pin is the SPI Data In pin, I2C Data pin and the 8-bit data bit 1 pin. This is a 3V logic level input pin when used with I2C/SPI, and an input/output when used in 8-bit.
- Pins #9-14 are Data2-7 - Used for 8-bit mode. These is a 3V input/output when used in 8-bit. Do not connect if using SPI or I2C
- Pin #15 is CS - the chip select pin. This is a 3V logic level input pin and is used for both SPI and 8-bit connections
- Pin #16 is RESET - the reset pin. This is a 3V logic level input pin and is used for I2C, SPI and 8-bit connections

Remaining Pins

- Pins #17-19 are not connected, do not use
- Pin #20 is the 'frame ground' pin and is connected to the metal case around the OLED, you can connect to ground or leave floating.

Assembly

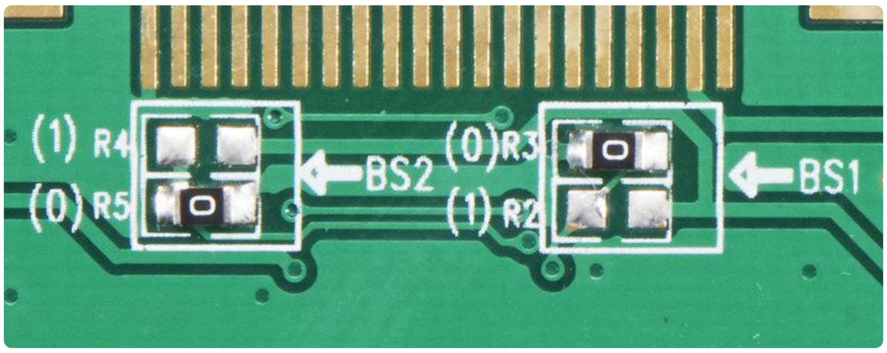
Changing "modes"

These modules can be used in SPI or 8-Bit mode. Somewhat annoyingly, the only way to switch modes is to desolder/solder jumpers on the back of the modules.

SPI Mode

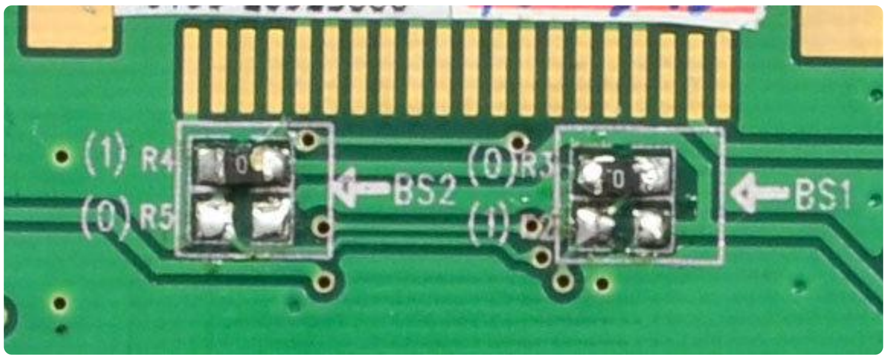
This is the mode you likely want to be in. Your module probably came with this setting by default.

Make sure the R5 and R3 resistors are soldered in and the R2 and R4 spots are blank



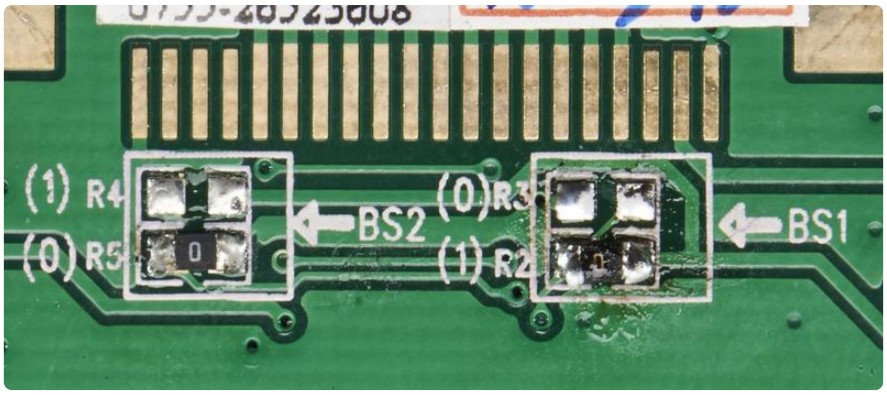
8-Bit "6800" mode

Make sure the R3 and R4 resistors are in place and the R2 and R5 are missing. You'll need to remove the R5 resistor to move it, by heating up the resistor with a soldering iron and maybe even melting a little solder on.



I2C mode

Make sure the R2 and R5 resistors are in place and the R3 and R4 are missing. You'll need to remove the R3 resistor to move it, by heating up the resistor with a soldering iron and maybe even melting a little solder on.



(If you solder and re-solder the resistors 10 times like I did while figuring out all the setting and wiring, you'll get a kinda messy/fluxy look like the above! In that case you can clean up with some IPA wipes. Or just revel in your punk soldering job)

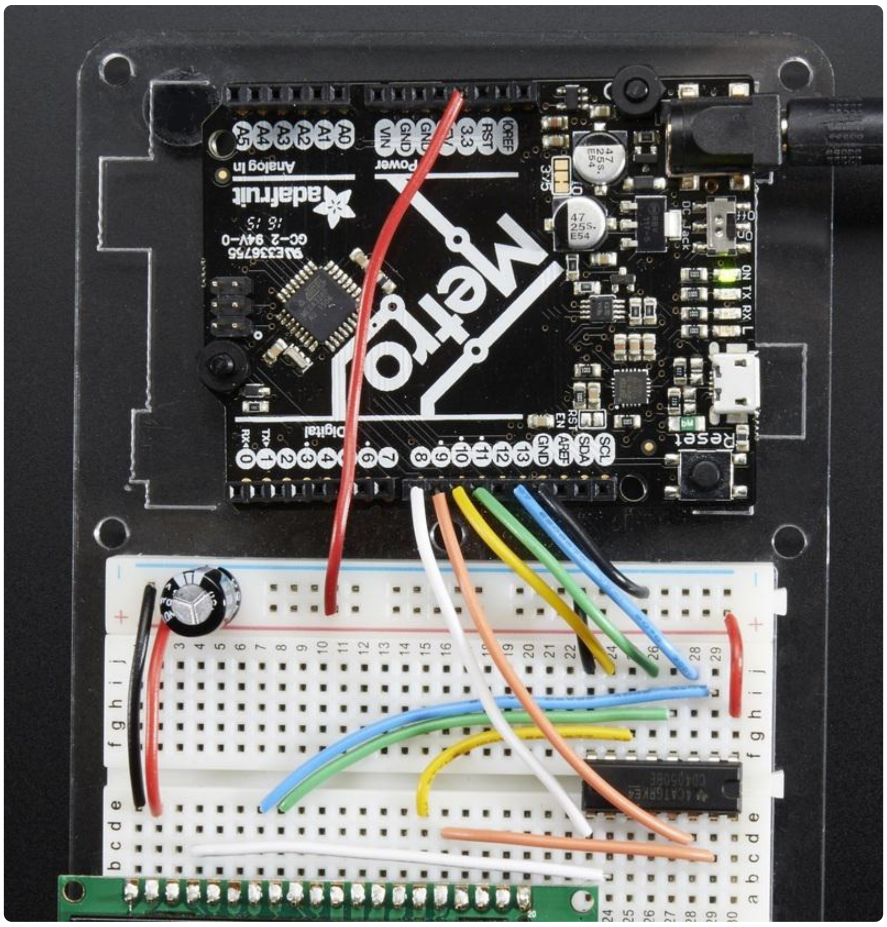
Arduino Wiring & Test

We will demonstrate using this display with an Arduino UNO compatible. If you are using a 3V logic device you can skip the level shifter and connect direct from the microcontroller to display. You can also use another kind of level shifter if you like.

Any microcontroller with I2C + 1 pin or 4 or 5 pins can be used, but we recommend testing it out with an UNO before you try a different processor.

SPI Wiring

NOTE: This example shows usage with a 5V board which requires level shifting.



Don't forget you have to set the display to SPI mode, see the Assembly step on how to do that!

Since this is a SPI-capable display, we can use hardware or 'software' SPI. To make wiring identical on all Arduinos, we'll begin with 'software' SPI. The following pins should be used:

- Connect Pin #1 to common power/data ground line (black wires)
- Connect Pin #2 to the 3V power supply on your Arduino. (red wires)
- Skip pin #3
- Connect Pin #4 (DC) to digital #8 via the level shifter (white wires) any pin can be used later
- Connect Pin #7 (SCLK) to digital #13 via the level shifter (blue wires) any pin can be used later
- Connect Pin #8 (DIN) to digital #11 via the level shifter (green wires) any pin can be used later
- Skip pins #9-14
- Connect Pin #15 (CS) to digital #10 via the level shifter (yellow wires) any pin can be used later

- Connect Pin #16 (RST) to digital #9 via the level shifter (orange wires) any pin can be used later

Later on, once we get it working, we can adjust the library to use hardware SPI if you desire, or change the pins to any others.

Level Shifter Wiring

You will also want to power the HC4050 level shifter by connecting pin #1 to 3V (the red wire) and pin #8 to ground (the black wire)

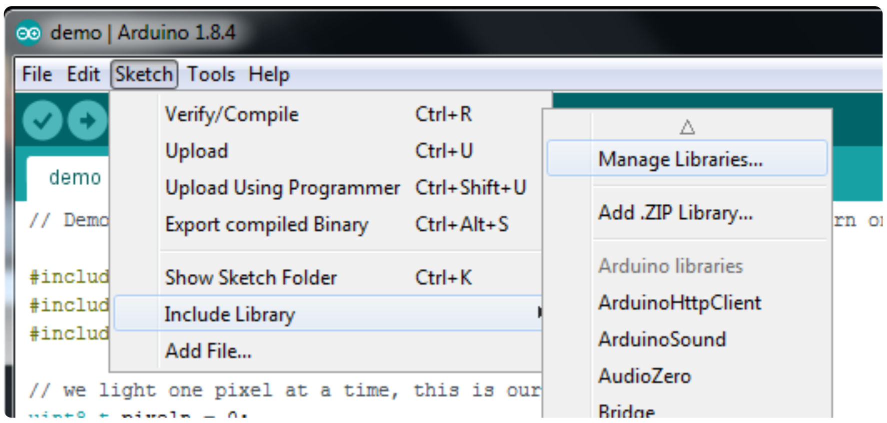
3.3V Capacitor

We also include a 220uF capacitor with your order because we noticed that the 3V line can fluctuate a lot when powered via an Arduino's 3.3V regulator. We really recommend installing it. Clip the leads on this capacitor and connect the negatve pin to GND and the positive pin to 3V

Download Libraries

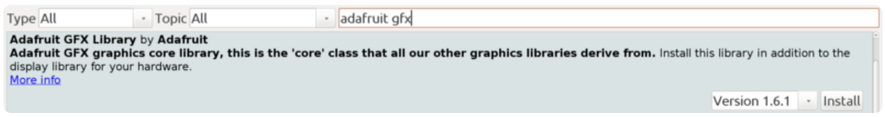
To begin reading sensor data, you will need to download Adafruit_SSD1305 () and Adafruit_GFX (). You can install these libraries via the Arduino library manager.

Open up the Arduino library manager:



```
// Demo
#includ
#includ
#includ
// we light one pixel at a time, this is our
```

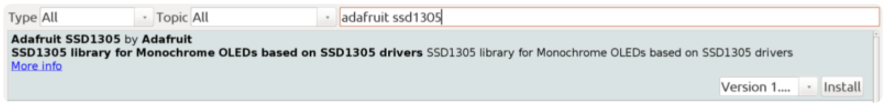
Search for the Adafruit GFX library and install it:



Adafruit GFX Library by **Adafruit**
Adafruit GFX graphics core library, this is the 'core' class that all our other graphics libraries derive from. Install this library in addition to the display library for your hardware.
More info

If using an older (pre-1.8.10) Arduino IDE, locate and install Adafruit_BusIO (newer versions do this one automatically).

Then, search for the Adafruit SSD1305 library and install it

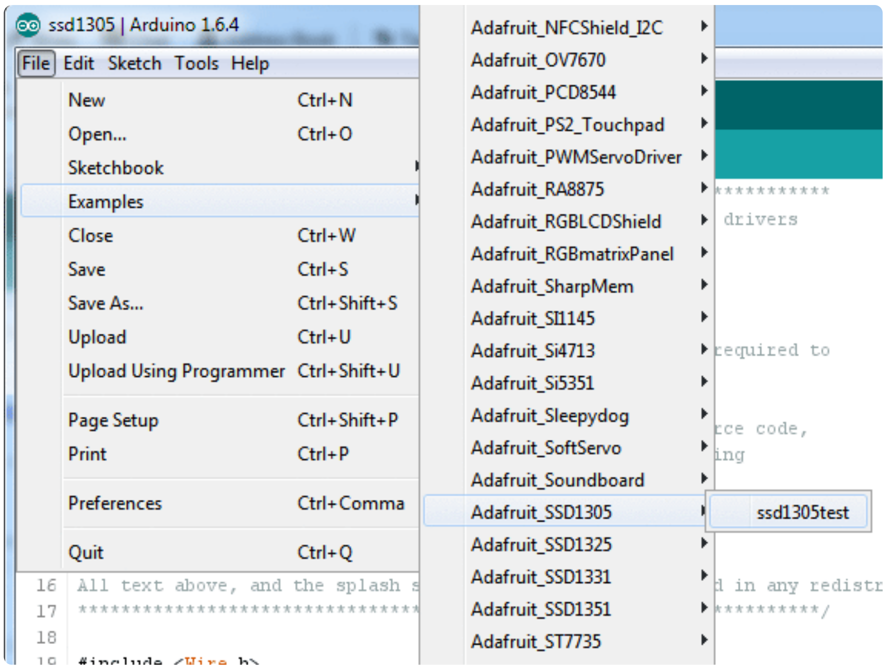


Adafruit SSD1305 by **Adafruit**
SSD1305 library for Monochrome OLEDs based on SSD1305 drivers SSD1305 library for Monochrome OLEDs based on SSD1305 drivers
More info

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

Running the Demo

After restarting the Arduino software, you should see a new example folder called Adafruit_SSD1305 and inside, an example called ssd1305test



Now upload the sketch to your Arduino. That's pretty much it! You should see immediate update of the display.

If nothing shows up at all, make sure you have your wiring correct, we have a diagram above you can use. Also, check that you converted the module to "SPI" mode (see the Assembly) step on how to do that

Adjust display size

The display size, in terms of width and height, is specified as the first two parameters passed in when creating the display instance. For example, for a display with width=128 and height=64:

```
Adafruit_SSD1305 display(128, 64, ...
```

Change these as needed for the display size being used.

Changing Pins

Now that you have it working, there's a few things you can do to change around the pins.

If you're using Hardware SPI, the CLOCK and MOSI pins are 'fixed' and cant be changed. But you can change to software SPI, which is a bit slower, and that lets you pick any pins you like. Find these lines:

```
      // If using software SPI, define CLK and MOSI
#define OLED_CLK 13
#define OLED_MOSI 11

// These are neede for both hardware &amp; softare SPI
#define OLED_CS 10
#define OLED_RESET 9
#define OLED_DC 8
```

Change those to whatever you like!

Using Hardware SPI

If you want a little more speed, you can 'upgrade' to Hardware SPI. Its a bit faster, maybe 2x faster to draw but requires you to use the hardware SPI pins.

- SPI CLK connects to SPI clock. On Arduino Uno/Duemilanove/328-based, thats Digital 13. On Mega's, its Digital 52 and on Leonardo/Due its ICSP-3 (See SPI Connections for more details ())

- SPI DATA IN connects to SPI MOSI. On Arduino Uno/Duemilanove/328-based, thats Digital 11. On Mega's, its Digital 51 and on Leonardo/Due its ICSP-4 (See SPI Connections for more details ())

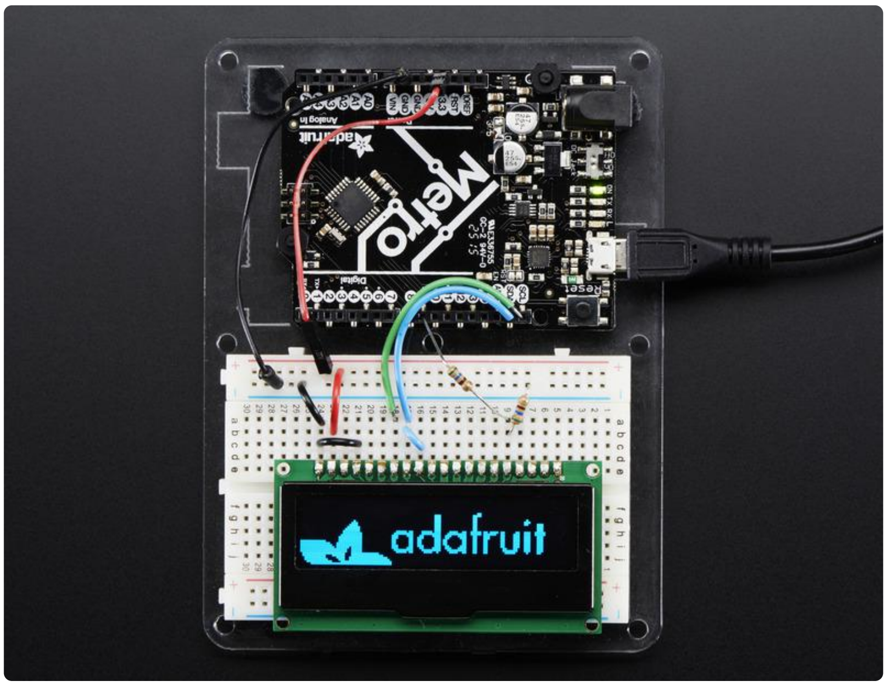
To enable hardware SPI, look for these lines:

```
// software SPI
//Adafruit_SSD1305 display(128, 64, OLED_MOSI, OLED_CLK, OLED_DC, OLED_RESET,
OLED_CS);
// hardware SPI - use 7Mhz (7000000UL) or lower because the screen is rated for
4MHz, or it will remain blank!
Adafruit_SSD1305 display(128, 64, &amp;SPI, OLED_DC, OLED_RESET, OLED_CS,
7000000UL);
```

Make sure the software SPI line is commented out and the hardware SPI line is uncommented.

I2C Wiring

It is also possible to use the display in I2C mode. Its a little slower but uses way fewer pins.



Don't forget you have to set the display to I2C mode, see the Assembly step on how to do that!

Unless you are using a Metro 328 you will need to add I2C pullups on SDA and SCL! Use two 10K (or so) resistors, each one connected from SDA & SCL to 3.3V

For I2C you will need to use the hardware I2C pins on your Arduino or microcontroller. The following pins should be used:

- Connect Pin #1 to common power/data ground line (black wires)
- Connect Pin #2 to the 3V power supply on your Arduino. (red wires)
- Skip pin #3
- Connect Pin #4 (DC & I2C Addr0) to ground (black wire) to set the I2C address to 0x3C. If this is tied to 3.3V, it will set the I2C address to 0x3D
- Connect Pin #7 (SCL) to Arduino SCL (green wire)
- Connect 10K resistor from SCL to 3.3V
- Connect Pin #8 (SDA) to Arduino SDA (blue wire)
- Connect 10K resistor from SDA to 3.3V
- Connect Pin #9 (SDA2) to Pin #8 (small blue wire)
- Skip pins #9-15
- Connect Pin #16 (RST) to digital #9 by using a resistive divider as shown, two resistors from 1K to 10K both the same value can be used. Any pin can be used later

While its ideal to use level shifters on the I2C pins, you can sorta get away with this on an arduino, because the I2C pins are open collector and there are very weak pullups on those two lines. If using with other I2C devices, we suggest using a 3V-logic arduino or an I2C-safe shifter. (http://adafru.it/757)

Later on, once we get it working, we can adjust the library to use hardware SPI if you desire, or change the pins to any others.

3.3V Capacitor

We also include a 220uF capacitor with your order because we noticed that the 3V line can fluctuate a lot when powered via an Arduino's 3.3V regulator. We really recommend installing it. Clip the leads on this capacitor and connect the negatve pin to GND and the positive pin to 3V

I2C code changes

In the test code, change the top area where you define the protocol used by commenting out the software and hardware SPI and uncommenting the I2C version

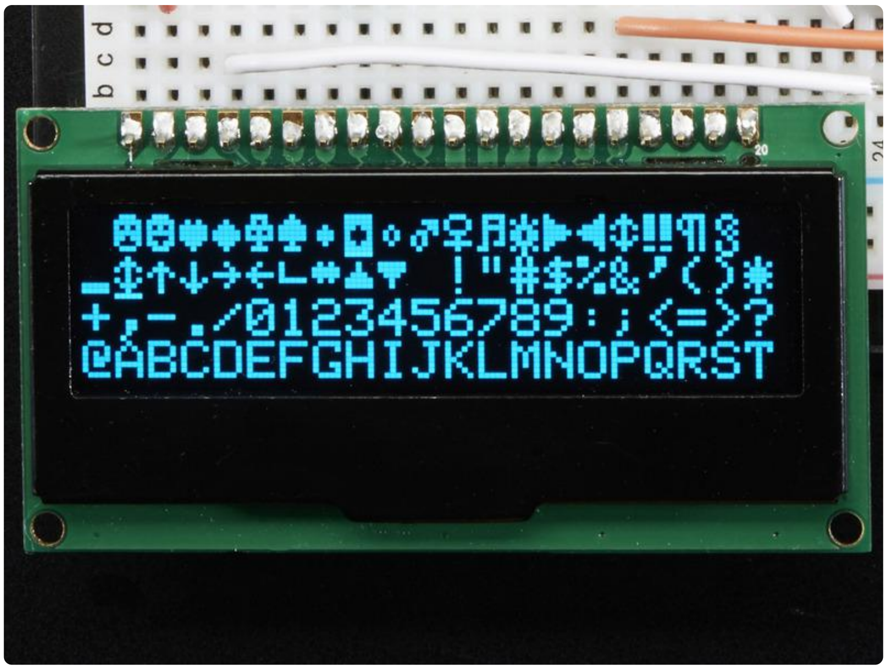
```
// software SPI
//Adafruit_SSD1305 display(128, 64, OLED_MOSI, OLED_CLK, OLED_DC, OLED_RESET,
OLED_CS);
// hardware SPI - use 7Mhz (7000000UL) or lower because the screen is rated for
4MHz, or it will remain blank!
//Adafruit_SSD1305 display(128, 64, &SPI, OLED_DC, OLED_RESET, OLED_CS, 7000000UL);

// I2C
Adafruit_SSD1305 display(128, 64, &Wire, OLED_RESET);
```

Everything else about the display is identical to SPI mode.

By default we use I2C address 0x3C which is what we get by connecting DC/A0 to ground. If you tie that pin to 3.3V instead, the address will be 0x3D and all you have to do is call display.begin(0x3D) to initialize with that address.

Using Adafruit GFX

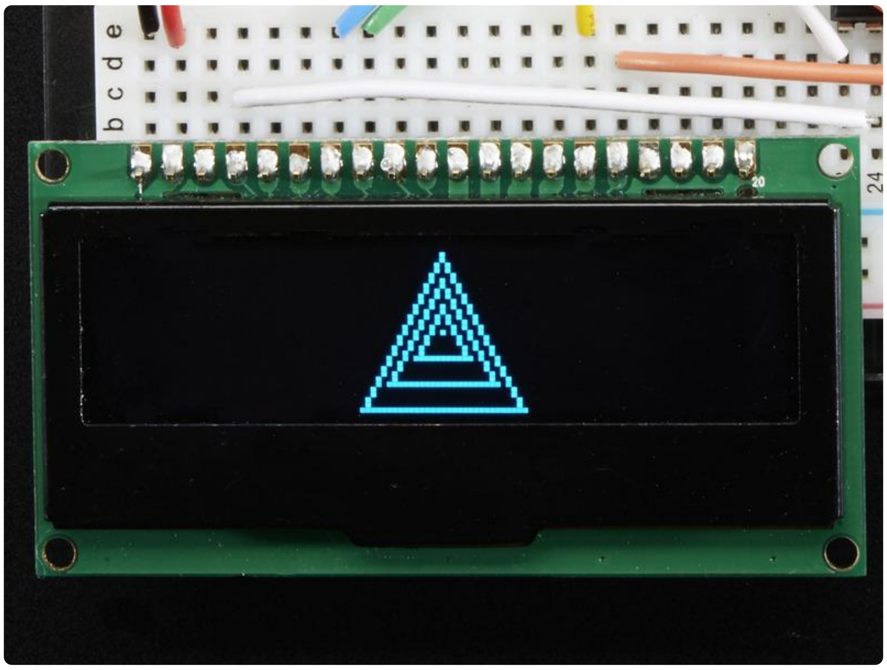


The Adafruit_GFX library for Arduino provides a common syntax and set of graphics functions for all of our TFT, LCD and OLED displays. This allows Arduino sketches to easily be adapted between display types with minimal fuss…and any new features, performance improvements and bug fixes will immediately apply across our complete offering of displays.

The GFX library is what lets you draw points, lines, rectangles, round-rects, triangles, text, etc.

Check out our detailed tutorial here http://learn.adafruit.com/adafruit-gfx-graphics-library () It covers the latest and greatest of the GFX library!

Since this is a 'buffered' display, dont forget to call the "display()" object function whenever you want to update the OLED. The entire display is drawn in one data burst, so this way you can put down a bunch of graphics and display it all at once.



CircuitPython Wiring

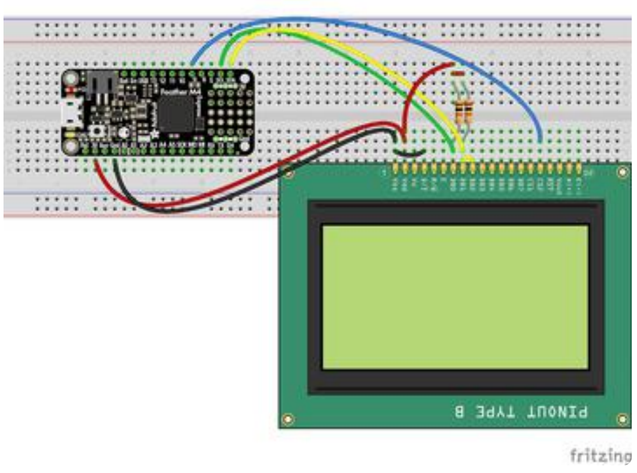
You can use this sensor with any CircuitPython microcontroller board or with a computer that has GPIO and Python thanks to Adafruit_Blinka, our CircuitPython-for-Python compatibility library ().

We'll cover how to wire the OLED to your CircuitPython microcontroller board. First assemble your OLED.

Connect the OLED to your microcontroller board as shown below.

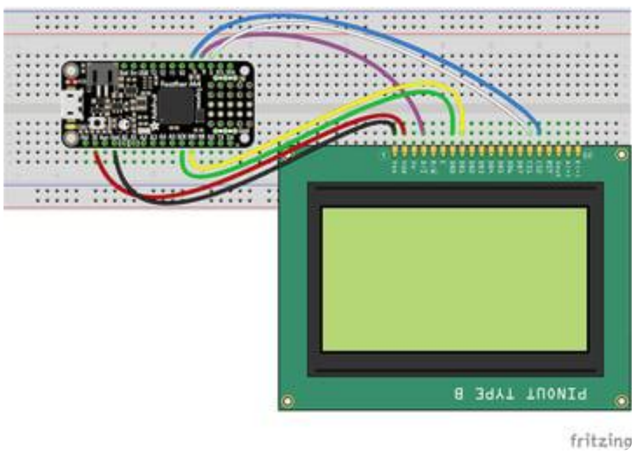
There's no SSD1305 or SSD1325 Large OLED Fritzing objects, so we sub'd a Graphic LCD in

Adafruit OLED Display I2C Wiring



OLED Pin #1 to Microcontroller GND
OLED Pin #2 to Microcontroller 3.3V
OLED Pin #4 to Microcontroller GND
OLED Pin #7 to Microcontroller SCL
10K resistor from SCL to 3.3V
OLED Pin #8 to Microcontroller SDA
OLED Pin #9 to Microcontroller SDA
10K resistor from SDA to 3.3V
OLED Pin #16 to Microcontroller D9

Adafruit OLED Display SPI Wiring



OLED Pin #1 to Microcontroller GND
OLED Pin #2 to Microcontroller 3.3V
OLED Pin #4 to Microcontroller D6
OLED Pin #7 to Microcontroller SCK
OLED Pin #8 to Microcontroller MOSI
OLED Pin #15 to Microcontroller D5
OLED Pin #16 to Microcontroller D9

Download the Fritzing Object

CircuitPython Setup

CircuitPython Installation of DisplayIO SSD1305 Library

To use the SSD1305 OLED with your Adafruit CircuitPython board you'll need to install the Adafruit CircuitPython DisplayIO SSD1305 () module on your board.

First make sure you are running the latest version 5.0 or later of Adafruit CircuitPython () for your board.

Next you'll need to install the necessary libraries to use the hardware--carefully follow the steps to find and install these libraries from Adafruit's CircuitPython library bundle (). Our CircuitPython starter guide has a great page on how to install the library bundle ().

If you choose, you can manually install the libraries individually on your board:

- adafruit_displayio_ssd1305

Before continuing make sure your board's lib folder or root filesystem has the adafruit_displayio_ssd1305.mpy file copied over.

Next connect to the board's serial REPL () so you are at the CircuitPython >>> prompt.

Code Example Additional Libraries

For the Code Example, you will need an additional library. We decided to make use of a library so the code didn't get overly complicated.

Adafruit_CircuitPython_Display_Text

Go ahead and install this in the same manner as the driver library by copying the adafruit_display_text folder over to the lib folder on your CircuitPython device.

CircuitPython Usage

Displayio is only available on express board due to the smaller memory size on non-express boards.

It's easy to use OLEDs with Python and the Adafruit CircuitPython DisplayIO SSD1305 () module. This module allows you to easily write Python code to control the display.

To demonstrate the usage, we'll initialize the library and use Python code to control the OLED from the board's Python REPL.

I2C Initialization

If your display is connected to the board using I2C you'll first need to initialize the I2C bus. First import the necessary modules:

```
import board
```

Now for run this command to create the I2C instance using the default SCL and SDA pins (which will be marked on the board's pins if using a Feather or similar Adafruit board):

```
i2c = board.I2C()
```

After initializing the I2C interface for your firmware as described above, you can create an instance of the I2CDisplay bus:

```
import displayio
import adafruit_displayio_ssd1305
display_bus = displayio.I2CDisplay(i2c, device_address=0x3c, reset=board.D9)
```

Finally, you can pass the display_bus in and create an instance of the SSD1305 I2C driver by running:

```
display = adafruit_displayio_ssd1305.SSD1305(display_bus, width=128, height=64)
```

Now you should be seeing an image of the REPL. Note that the last two parameters to the `SSD1305` class initializer are the width and height of the display in pixels. Be sure to use the right values for the display you're using!

Changing the I2C address

If you connect Pin #4 of the OLED to +3V instead of Ground the I2C address will be different different (`0x3d`):

```
display_bus = displayio.I2CDisplay(i2c, device_address=0x3d, reset=board.D9)
display = adafruit_displayio_ssd1305.SSD1305(display_bus, width=128, height=64)
```

At this point the I2C bus and display are initialized. Skip down to the example code section.

SPI Initialization

If your display is connected to the board using SPI you'll first need to initialize the SPI bus.

If you're using a microcontroller board, run the following commands:

```
import board
import displayio
import adafruit_displayio_ssd1305

displayio.release_displays()

spi = board.SPI()
tft_cs = board.D5
tft_dc = board.D6
tft_reset = board.D9

display_bus = displayio.FourWire(spi, command=tft_dc, chip_select=tft_cs,
                                 reset=tft_reset, baudrate=1000000)
display = adafruit_displayio_ssd1305.SSD1305(display_bus, width=128, height=64)
```

The parameters to the FourWire initializer are the pins connected to the display's DC, CS, and reset. Because we are using keyword arguments, they can be in any position. Again make sure to use the right pin names as you have wired up to your board!

Note that the last two parameters to the `SSD1305` class initializer are the width and h eight of the display in pixels. Be sure to use the right values for the display you're using!

Example Code

```
# SPDX-FileCopyrightText: 2021 ladyada for Adafruit Industries
# SPDX-License-Identifier: MIT

"""
This test will initialize the display using displayio and draw a solid white
background, a smaller black rectangle, and some white text.
"""

import board
import displayio
import terminalio
from adafruit_display_text import label
import adafruit_displayio_ssd1305

displayio.release_displays()

# Reset is usedfor both SPI and I2C
oled_reset = board.D9
```

```
# Use for SPI
spi = board.SPI()
oled_cs = board.D5
oled_dc = board.D6
display_bus = displayio.FourWire(
    spi, command=oled_dc, chip_select=oled_cs, baudrate=1000000, reset=oled_reset
)

# Use for I2C
# i2c = board.I2C()  # uses board.SCL and board.SDA
# i2c = board.STEMMA_I2C()  # For using the built-in STEMMA QT connector on a
microcontroller
# display_bus = displayio.I2CDisplay(i2c, device_address=0x3c, reset=oled_reset)

WIDTH = 128
HEIGHT = 64  # Change to 32 if needed
BORDER = 8
FONTSCALE = 1

display = adafruit_displayio_ssd1305.SSD1305(display_bus, width=WIDTH,
height=HEIGHT)

# Make the display context
splash = displayio.Group()
display.show(splash)

color_bitmap = displayio.Bitmap(display.width, display.height, 1)
color_palette = displayio.Palette(1)
color_palette[0] = 0xFFFFFF  # White

bg_sprite = displayio.TileGrid(color_bitmap, pixel_shader=color_palette, x=0, y=0)
splash.append(bg_sprite)

# Draw a smaller inner rectangle
inner_bitmap = displayio.Bitmap(
    display.width - BORDER * 2, display.height - BORDER * 2, 1
)
inner_palette = displayio.Palette(1)
inner_palette[0] = 0x000000  # Black
inner_sprite = displayio.TileGrid(
    inner_bitmap, pixel_shader=inner_palette, x=BORDER, y=BORDER
)
splash.append(inner_sprite)

# Draw a label
text = "Hello World!"
text_area = label.Label(terminalio.FONT, text=text, color=0xFFFFFF)
text_width = text_area.bounding_box[2] * FONTSCALE
text_group = displayio.Group(
    scale=FONTSCALE,
    x=display.width // 2 - text_width // 2,
    y=display.height // 2,
)
text_group.append(text_area)  # Subgroup for text scaling
splash.append(text_group)

while True:
    pass
```

Let's take a look at the sections of code one by one. We start by importing the `board` so that we can initialize SPI, `displayio`,`terminalio` for the font, a `label`, and the `adafruit_displayio_ssd1305` driver.

```
import board
import displayio
import terminalio
from adafruit_display_text import label
import adafruit_displayio_ssd1305
```

Next we release any previously used displays. This is important because if the microprocessor is reset, the display pins are not automatically released and this makes them available for use again.

```
displayio.release_displays()
```

Here we set `oled_reset` to board.D9 which will be used with either SPI or I2C. If your board is wired differently, be sure to change it to match your wiring.

```
oled_reset = board.D9
```

If you're using SPI, you would use this section of code. We set the SPI object to the board's SPI with the easy shortcut function `board.SPI()`. By using this function, it finds the SPI module and initializes using the default SPI parameters. We set the OLED's CS (Chip Select), and DC (Data/Command) pins. We also set the display bus to FourWire which makes use of the SPI bus. The SSD1305 needs to be slowed down to 1MHz, so we pass in the additional `baudrate` parameter. We also pass `oled_reset` as the reset pin. If this differs for you, you could change it here.

```
spi = board.SPI()
oled_cs = board.D5
oled_dc = board.D6
display_bus = displayio.FourWire(spi, command=oled_dc, chip_select=oled_cs,
                                 baudrate=1000000, reset=oled_reset)
```

If you're using I2C, you would use this section of code. We set the I2C object to the board's I2C with the easy shortcut function `board.I2C()`. By using this function, it finds the SPI module and initializes using the default SPI parameters. We also set the display bus to I2CDisplay which makes use of the I2C bus.

```
# Use for I2C
i2c = board.I2C()
display_bus = displayio.I2CDisplay(i2c, device_address=0x3c, reset=oled_reset)
```

In order to make it easy to change display sizes, we'll define a few variables in one spot here. We have `WIDTH`, which is the display width, `HEIGHT`, which is the display height and `BORDER`, which we will explain a little further below. `FONTSCALE` will be the multiplier for the font size. If your display is something different than these numbers, change them to the correct setting. For instance, you may want try changing the border size to 5 if you have a 128x32 display.

```
WIDTH = 128
HEIGHT = 64     # Change to 32 if needed
BORDER = 8
FONTSCALE = 1
```

Finally, we initialize the driver with a width of the `WIDTH` variable and a height of the `HEIGHT` variable. If we stopped at this point and ran the code, we would have a terminal that we could type at and have the screen update.

```
display = adafruit_displayio_ssd1305.SSD1305(display_bus, width=WIDTH,
height=HEIGHT)
```

Next we create a background splash image. We do this by creating a group that we can add elements to and adding that group to the display. In this example, we are limiting the maximum number of elements to 10, but this can be increased if you would like. The display will automatically handle updating the group.

```
splash = displayio.Group(max_size=10)
display.show(splash)
```

Next we create a Bitmap that is the full width and height of the display. The Bitmap is like a canvas that we can draw on. In this case we are creating the Bitmap to be the same size as the screen, but only have one color. Although the Bitmaps can handle up to 256 different colors, the display is monochrome so we only need one. We create a Palette with one color and set that color to `0xFFFFFF` which happens to be white. If were to place a different color here, `displayio` handles color conversion automatically, so it may end up black or white depending on the calculation.

```
color_bitmap = displayio.Bitmap(WIDTH, HEIGHT, 1)
color_palette = displayio.Palette(1)
color_palette[0] = 0xFFFFFF # White
```

With all those pieces in place, we create a TileGrid by passing the bitmap and palette and draw it at `(0, 0)` which represents the display's upper left.

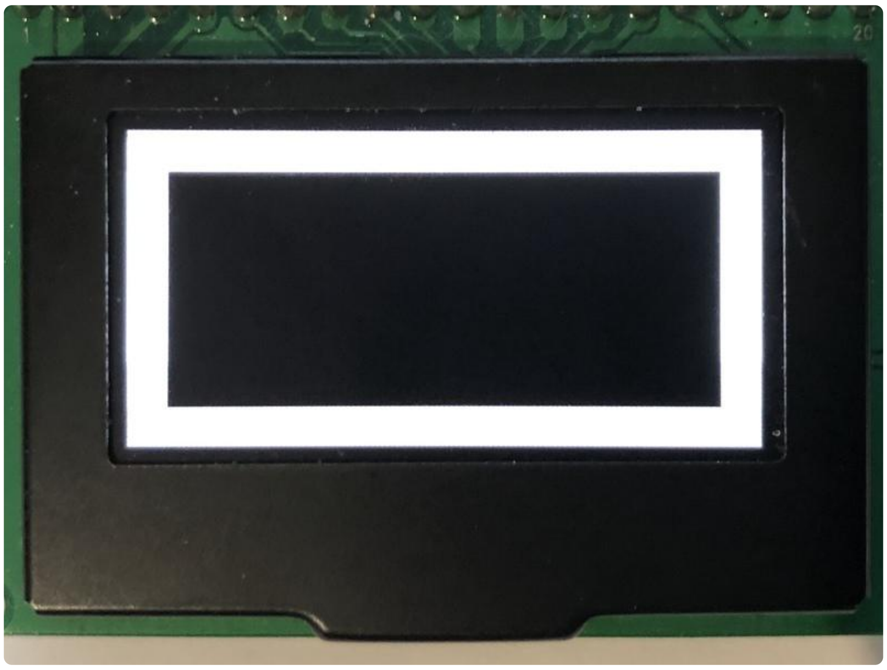
```
bg_sprite = displayio.TileGrid(color_bitmap,
                               pixel_shader=color_palette,
                               x=0, y=0)
splash.append(bg_sprite)
```

Next we will create a smaller black rectangle. The easiest way to do this is to create a new bitmap that is a little smaller than the full screen with a single color of `0x000000`, which is black, and place it in a specific location. In this case, we will create a bitmap that is 5 pixels smaller on each side. This is where the BORDER variable comes into use. It makes calculating the size of the second rectangle much easier. The screen we're using here is 128x64 and we have the BORDER set to 8 , so we'll want to subtract 16 from each of those numbers.

We'll also want to place it at the position `(8, 8)` so that it ends up centered.

```
# Draw a smaller inner rectangle
inner_bitmap = displayio.Bitmap(display.width - BORDER * 2, display.height - BORDER
* 2, 1)
inner_palette = displayio.Palette(1)
inner_palette[0] = 0x000000 # Black
inner_sprite = displayio.TileGrid(inner_bitmap,
                                  pixel_shader=inner_palette,
                                  x=BORDER, y=BORDER)
splash.append(inner_sprite)
```

Since we are adding this after the first square, it's automatically drawn on top. Here's what it looks like now.



Next let's add a label that says "Hello World!" on top of that. We're going to use the built-in Terminal Font and scale it up by a factor of two, which is what we have `FONTSCALE` set to. To scale the label only, we will make use of a subgroup, which we will then add to the main group.

We create the label first so that we can get the width of the bounding box and multiply it by the `FONTSCALE`. This gives us the actual with of the text.

Labels are automatically centered vertically, so we'll place it at half the display height for the Y coordinate, and we calculate the X coordinate to horizontally center the label. We use the `//` operator to divide because we want a whole number returned and it's an easy way to round it. Let's go with some white text, so we'll pass it a value of `0xFFFFFF`.

```
# Draw a label
text = "Hello World!"
text_area = label.Label(terminalio.FONT, text=text, color=0xFFFFFF)
text_width = text_area.bounding_box[2] * FONTSCALE
text_group = displayio.Group(max_size=10, scale=FONTSCALE, x=display.width // 2 -
text_width // 2,
                             y=display.height // 2)
text_group.append(text_area) # Subgroup for text scaling
splash.append(text_group)
```

Finally, we place an infinite loop at the end so that the graphics screen remains in place and isn't replaced by a terminal.

```
while True:
    pass
```

If you have the 2.3" 128x32 OLED Display, here's what the final output looks like with the height set to 32 and the border size set to 5:

Where to go from here

Be sure to check out this excellent guide to CircuitPython Display Support Using displayio ()

Python Wiring

It's easy to use OLEDs with Python and the Adafruit CircuitPython SSD1305 () module. This module allows you to easily write Python code to control the display.

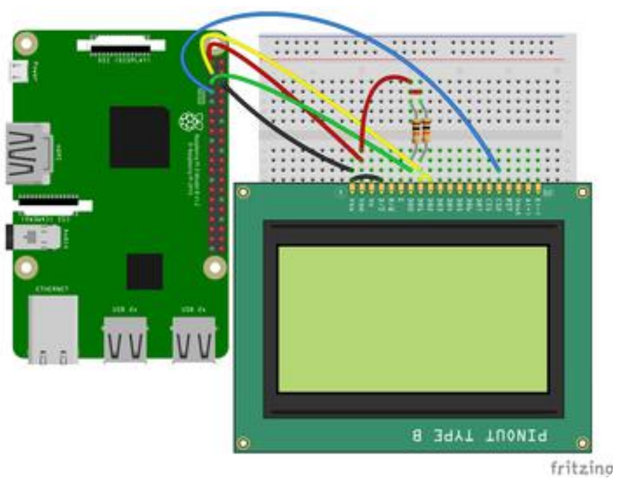
We'll cover how to wire the OLED to your Raspberry Pi. First assemble your OLED.

Since there's dozens of Linux computers/boards you can use we will show wiring for Raspberry Pi. For other platforms, please visit the guide for CircuitPython on Linux to see whether your platform is supported ().

Connect the OLED as shown below to your Raspberry Pi.

There's no SSD1305 Large OLED Fritzing object, so we sub'd a Graphic LCD in

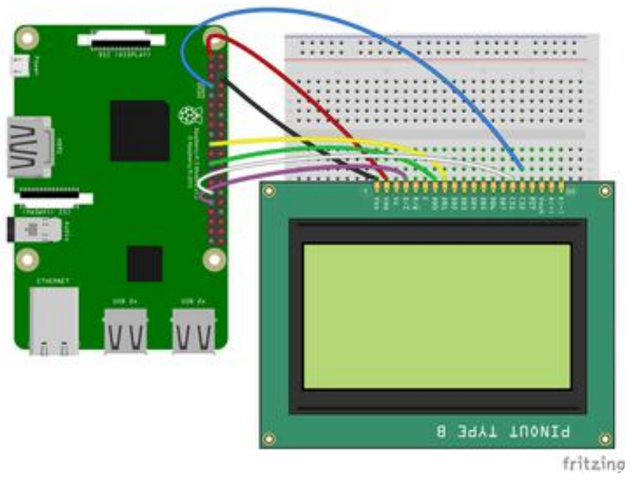
Adafruit OLED Display I2C Wiring



- OLED Pin #1 to Raspberry Pi GND
- OLED Pin #2 to Raspberry Pi 3.3V
- OLED Pin #4 to Raspberry Pi GND
- OLED Pin #7 to Raspberry Pi SCL
- 10K resistor from SCL to 3.3V
- OLED Pin #8 to Raspberry Pi SDA
- OLED Pin #9 to Raspberry Pi SDA
- 10K resistor from SDA to 3.3V
- OLED Pin #16 to Raspberry Pi GPIO 4

Download the Fritzing Object

Adafruit OLED Display SPI Wiring



OLED Pin #1 to Raspberry Pi GND
OLED Pin #2 to Raspberry Pi 3.3V
OLED Pin #4 to Raspberry Pi GPIO 6
OLED Pin #7 to Raspberry Pi SCK
OLED Pin #8 to Raspberry Pi MOSI
OLED Pin #15 to Raspberry Pi GPIO 5
OLED Pin #16 to Raspberry Pi GPIO 4

Download the Fritzing Object

Python Setup

You'll need to install the Adafruit_Blinka library that provides the CircuitPython support in Python. This may also require enabling I2C on your platform and verifying you are running Python 3. Since each platform is a little different, and Linux changes often, please visit the CircuitPython on Linux guide to get your computer ready ()!

Python Installation of SSD1305 Library

Once that's done, from your command line run the following command:

- `pip3 install adafruit-circuitpython-ssd1305`

If your default Python is version 3 you may need to run 'pip' instead. Just make sure you aren't trying to use CircuitPython on Python 2.x, it isn't supported!

If that complains about pip3 not being installed, then run this first to install it:

- `sudo apt-get install python3-pip`

Pillow Library

We also need PIL, the Python Imaging Library, to allow using text with custom fonts. There are several system libraries that PIL relies on, so installing via a package manager is the easiest way to bring in everything:

- `sudo apt-get install python3-pil`

That's it. You should be ready to go.

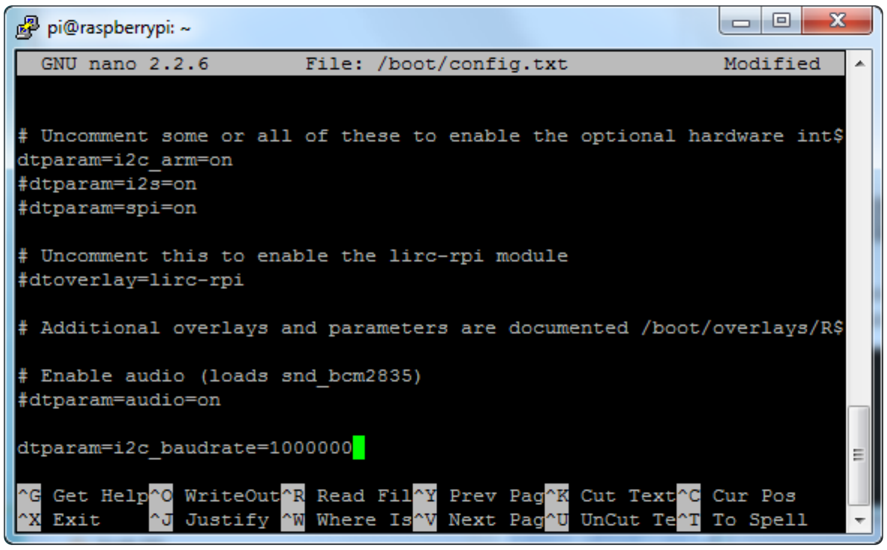
Speeding up an I2C Display on Raspberry Pi

For the best performance, especially if you are doing fast animations, you'll want to tweak the I2C core to run at 1MHz. By default it may be 100KHz or 400KHz

To do this edit the config with `sudo nano /boot/config.txt`

and add to the end of the file

`dtparam=i2c_baudrate=1000000`



```
  GNU nano 2.2.6            File: /boot/config.txt                    Modified


# Uncomment some or all of these to enable the optional hardware int$
dtparam=i2c_arm=on
#dtparam=i2s=on
#dtparam=spi=on

# Uncomment this to enable the lirc-rpi module
#dtoverlay=lirc-rpi

# Additional overlays and parameters are documented /boot/overlays/R$

# Enable audio (loads snd_bcm2835)
#dtparam=audio=on

dtparam=i2c_baudrate=1000000

^G Get Help^O WriteOut^R Read Fil^Y Prev Pag^K Cut Text^C Cur Pos
^X Exit    ^J Justify ^W Where Is^V Next Pag^U UnCut Te^T To Spell
```

reboot to 'set' the change.

Python Usage

It's easy to use OLEDs with Python and the Adafruit CircuitPython SSD1305 () module. This module allows you to easily write Python code to control the display.

You can use this sensor with any computer that has GPIO and Python thanks to Adafruit_Blinka, our CircuitPython-for-Python compatibility library ().

To demonstrate the usage, we'll initialize the library and use Python code to control the OLED from the board's Python REPL.

Since we are running full CPython on our Linux/computer, we can take advantage of the powerful Pillow image drawing library to handle text, shapes, graphics, etc. Pillow is a gold standard in image and graphics handling, you can read about all it can do here ().

I2C Initialization

If your display is connected to the board using I2C you'll first need to initialize the I2C bus. First import the necessary modules:

```
import board
import busio
```

Now for either board run this command to create the I2C instance using the default SCL and SDA pins of your I2C host:

```
i2c = busio.I2C(board.SCL, board.SDA)
```

After initializing the I2C interface for your firmware as described above you can create an instance of the SSD1305 I2C driver by running:

```
import adafruit_ssd1305
oled = adafruit_ssd1305.SSD1305_I2C(128, 64, i2c)
```

Note that the first two parameters to the `SSD1305_I2C` class initializer are the width and height of the display in pixels. Be sure to use the right values for the display you're using!

Changing the I2C address

If you connect Pin #4 of the OLED to +3V instead of Ground the I2C address will be different different (`0x3d`):

```
oled = adafruit_ssd1305.SSD1305_I2C(128, 64, i2c, addr=0x3d)
```

Adding hardware reset pin

If you have a reset pin, which may be required if your OLED does not have an auto-reset chip, also pass in a reset pin like so:

```
import digitalio

reset_pin = digitalio.DigitalInOut(board.D4) # any pin!
oled = adafruit_ssd1305.SSD1305_I2C(128, 32, i2c, reset=reset_pin)
```

At this point the I2C bus and display are initialized. Skip down to the example code section.

SPI Initialization

If your display is connected to the board using SPI you'll first need to initialize the SPI bus:

```
import adafruit_ssd1305
import board
import busio
import digitalio

spi = busio.SPI(board.SCK, MOSI=board.MOSI)
reset_pin = digitalio.DigitalInOut(board.D4) # any pin!
cs_pin = digitalio.DigitalInOut(board.D5)    # any pin!
dc_pin = digitalio.DigitalInOut(board.D6)    # any pin!

oled = adafruit_ssd1305.SSD1305_SPI(128, 32, spi, dc_pin, reset_pin, cs_pin)
```

Note the first two parameters to the `SSD1305_SPI` class initializer are the width and h eight of the display in pixels. Be sure to use the right values for the display you're using!

The next parameters to the initializer are the pins connected to the display's DC, reset, and CS lines in that order. Again make sure to use the right pin names as you have wired up to your board!

Example Code

```
# SPDX-FileCopyrightText: 2021 ladyada for Adafruit Industries
# SPDX-License-Identifier: MIT

"""
This demo will fill the screen with white, draw a black box on top
and then print Hello World! in the center of the display

This example is for use on (Linux) computers that are using CPython with
Adafruit Blinka to support CircuitPython libraries. CircuitPython does
not support PIL/pillow (python imaging library)!
"""

import board
import digitalio
from PIL import Image, ImageDraw, ImageFont
import adafruit_ssd1305

# Define the Reset Pin
oled_reset = digitalio.DigitalInOut(board.D4)

# Change these
# to the right size for your display!
WIDTH = 128
HEIGHT = 64  # Change to 32 if needed
BORDER = 8

# Use for SPI
spi = board.SPI()
oled_cs = digitalio.DigitalInOut(board.D5)
oled_dc = digitalio.DigitalInOut(board.D6)
oled = adafruit_ssd1305.SSD1305_SPI(WIDTH, HEIGHT, spi, oled_dc, oled_reset,
oled_cs)

# Use for I2C.
# i2c = board.I2C()  # uses board.SCL and board.SDA
# i2c = board.STEMMA_I2C()  # For using the built-in STEMMA QT connector on a
microcontroller
# oled = adafruit_ssd1305.SSD1305_I2C(WIDTH, HEIGHT, i2c, addr=0x3c,
reset=oled_reset)

# Clear display.
oled.fill(0)
oled.show()

# Create blank image for drawing.
# Make sure to create image with mode '1' for 1-bit color.
image = Image.new("1", (oled.width, oled.height))

# Get drawing object to draw on image.
draw = ImageDraw.Draw(image)

# Draw a white background
draw.rectangle((0, 0, oled.width, oled.height), outline=255, fill=255)

# Draw a smaller inner rectangle
draw.rectangle(
    (BORDER, BORDER, oled.width - BORDER - 1, oled.height - BORDER - 1),
    outline=0,
    fill=0,
)

# Load default font.
font = ImageFont.load_default()
```

```
# Draw Some Text
text = "Hello World!"
(font_width, font_height) = font.getsize(text)
draw.text(
    (oled.width // 2 - font_width // 2, oled.height // 2 - font_height // 2),
    text,
    font=font,
    fill=255,
)

# Display image
oled.image(image)
oled.show()
```

Let's take a look at the sections of code one by one. We start by importing the `board` so that we can initialize SPI, `digitalio`, several `PIL` modules for Image Drawing, and the `adafruit_ssd1305` driver.

```
import board
import digitalio
from PIL import Image, ImageDraw, ImageFont
import adafruit_ssd1305
```

Next we define the reset line, which will be used for either SPI or I2C. If your OLED has auto-reset circuitry, you can set the `oled_reset` line to None

```
oled_reset = digitalio.DigitalInOut(board.D4)
```

In order to make it easy to change display sizes, we'll define a few variables in one spot here. We have the display width, the display height and the border size, which we will explain a little further below. If your display is something different than these numbers, change them to the correct setting. For instance, you may want try changing the border size to 5 if you have a 128x32 display.

```
WIDTH = 128
HEIGHT = 64     # Change to 32 if needed
BORDER = 8
```

If you're using I2C, you would use this section of code. We set the I2C object to the board's I2C with the easy shortcut function `board.I2C()`. By using this function, it finds the SPI module and initializes using the default SPI parameters. We also set up the oled with SSD1305_I2C which makes use of the I2C bus.

```
# Use for I2C.
i2c = board.I2C()
oled = adafruit_ssd1305.SSD1305_I2C(WIDTH, HEIGHT, i2c, addr=0x3c, reset=oled_reset)
```

If you're using SPI, you would use this section of code. We set the SPI object to the board's SPI with the easy shortcut function `board.SPI()`. By using this function, it finds the SPI module and initializes using the default SPI parameters. We set the OLED's CS (Chip Select), and DC (Data/Command) pins. We also set up the OLED with SSD1305_SPI which makes use of the SPI bus.

```
# Use for SPI
spi = board.SPI()
oled_cs = digitalio.DigitalInOut(board.D5)
oled_dc = digitalio.DigitalInOut(board.D6)
oled = adafruit_ssd1305.SSD1305_SPI(WIDTH, HEIGHT, spi, oled_dc, oled_reset,
oled_cs)
```

Next we clear the display in case it was initialized with any random artifact data.

```
# Clear display.
oled.fill(0)
oled.show()
```

Next, we need to initialize PIL to create a blank image to draw on. Think of it as a virtual canvas. Since this is a monochrome display, we set it up for 1-bit color, meaning a pixel is either white or black. We can make use of the OLED's width and height properties as well. Optionally, we could have used our WIDTH and HEIGHT variables.

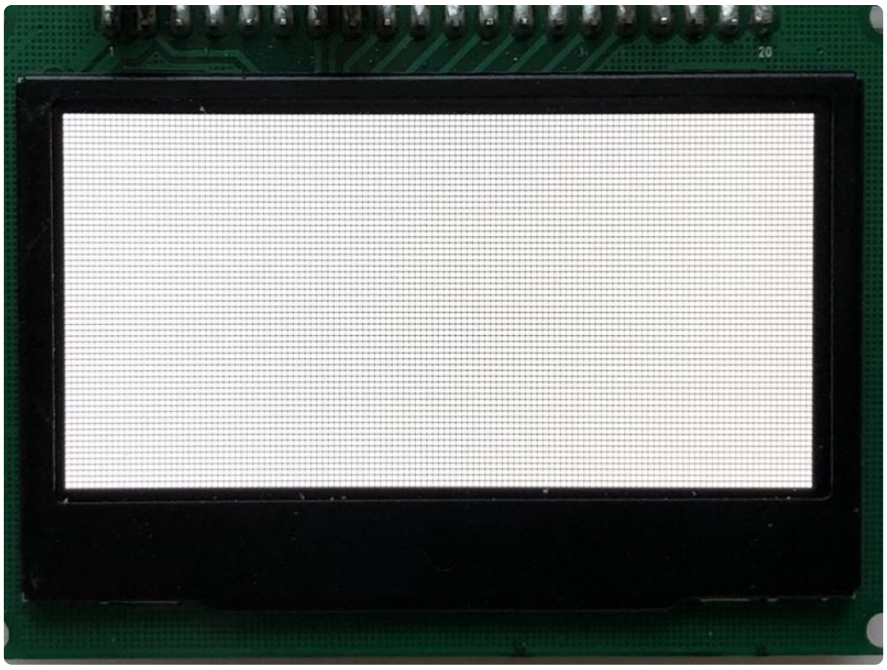
```
# Create blank image for drawing.
# Make sure to create image with mode '1' for 1-bit color.
image = Image.new('1', (oled.width, oled.height))

# Get drawing object to draw on image.
draw = ImageDraw.Draw(image)
```

Now we start the actual drawing. Here we are telling it we want to draw a rectangle from `(0,0)`, which is the upper left, to the full width and height of the display. We want it both filled in and having an outline of white, so we pass 255 for both numbers.

```
# Draw a white background
draw.rectangle((0, 0, oled.width, oled.height), outline=255, fill=255)
```

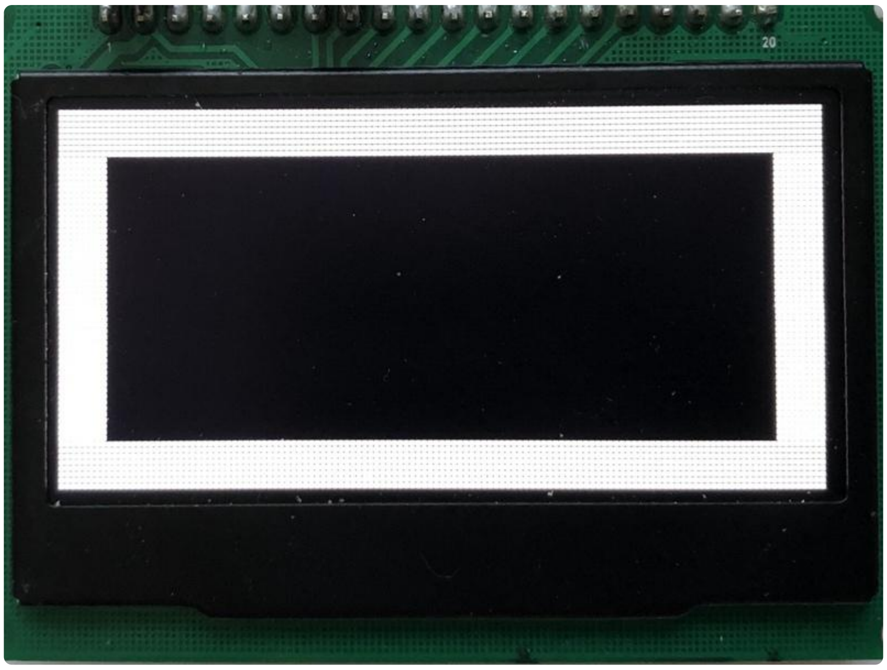
If we ran the code now, it would still show a blank display because we haven't told python to use our virtual canvas yet. You can skip to the end if you would like to see how to do that. This is what our canvas currently looks like in memory.



Next we will create a smaller black rectangle. The easiest way to do this is to draw another rectangle a little smaller than the full screen with no fill or outline and place it in a specific location. In this case, we will create a rectangle that is 5 pixels smaller on each side. This is where the BORDER variable comes into use. It makes calculating the size of the second rectangle much easier. We want the starting coordinate, which consists of the first two parameters, to be our BORDER value. Then for the next two parameters, which are our ending coordinates, we want to subtract our border value from the width and height. Also, because this is a zero-based coordinate system, we also need to subtract 1 from each number.

```
# Draw a smaller inner rectangle
draw.rectangle((BORDER, BORDER, oled.width - BORDER - 1, oled.height - BORDER - 1),
               outline=0, fill=0)
```

Here's what our virtual canvas looks like in memory.



Now drawing text with PIL is pretty straightforward. First we start by setting the font to the default system text. After that we define our text and get the size of the text. We're grabbing the size that it would render at so that we can calculate the center position. Finally, we take the font size and screen size to calculate the position we want to draw the text at and it appears in the center of the screen.

```
# Load default font.
font = ImageFont.load_default()

# Draw Some Text
text = "Hello World!"
(font_width, font_height) = font.getsize(text)
draw.text((oled.width//2 - font_width//2, oled.height//2 - font_height//2),
          text, font=font, fill=255)
```

Finally, we need to display our virtual canvas to the OLED and we do that with 2 commands. First we set the image to the screen, then we tell it to show the image.

```
# Display image
oled.image(image)
oled.show()
```

Don't forget you MUST call oled.image(image) and oled.show() to actually display the graphics. The OLED takes a while to draw so cluster all your drawing functions into the buffer (fast) and then display them once to the oled (slow)

Here's what the final output should look like.

If you have the 2.3" 128x32 OLED Display, here's what the final output looks like with the height set to 32 and the border size set to 5:

F.A.Q.

How come sometimes I see banding or dim areas on the OLED?

These OLEDs are passively drawn, which means that each line is lit at once. These displays are fairly inexpensive and simple, but as a tradeoff the built in boost converter has to drive all the OLED pixels at once. If you have a line with almost all the pixels lit it wont be as bright as a line with only 50% or less lit up.

The display works, because I can see the splash screen, but when I draw to the display nothing appears!

Don't forget you must call .display() to actually write the display data to the display. Unlike many of our TFTs, the entire display must be written at once so you should print all your text and draw all your squares, then call display()

How do I get rid of the splash screen?

Open up Adafruit_SSD1305.cpp in the libraries/Adafruit_SSD1305 folder and find these lines

```
static uint8_t buffer[SSD1305_LCDHEIGHT * SSD1305_LCDWIDTH / 8] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
....
0x00, 0x01, 0x00, 0x01, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x01, 0x01, 0x01,
0x01, 0x01
};
```

and delete everything after `static uint8_t buffer[SSD1305_LCDHEIGHT * SSD1305_LCDWIDTH / 8] = {`
and before `};`

Downloads

Datasheets

- Datasheet for the SSD1305, the passive OLED driver chip in the module () this is the chip in the module that converts SPI/8-bit commands to OLED control signals
- Mechanical specifications for the OLED display itself ()