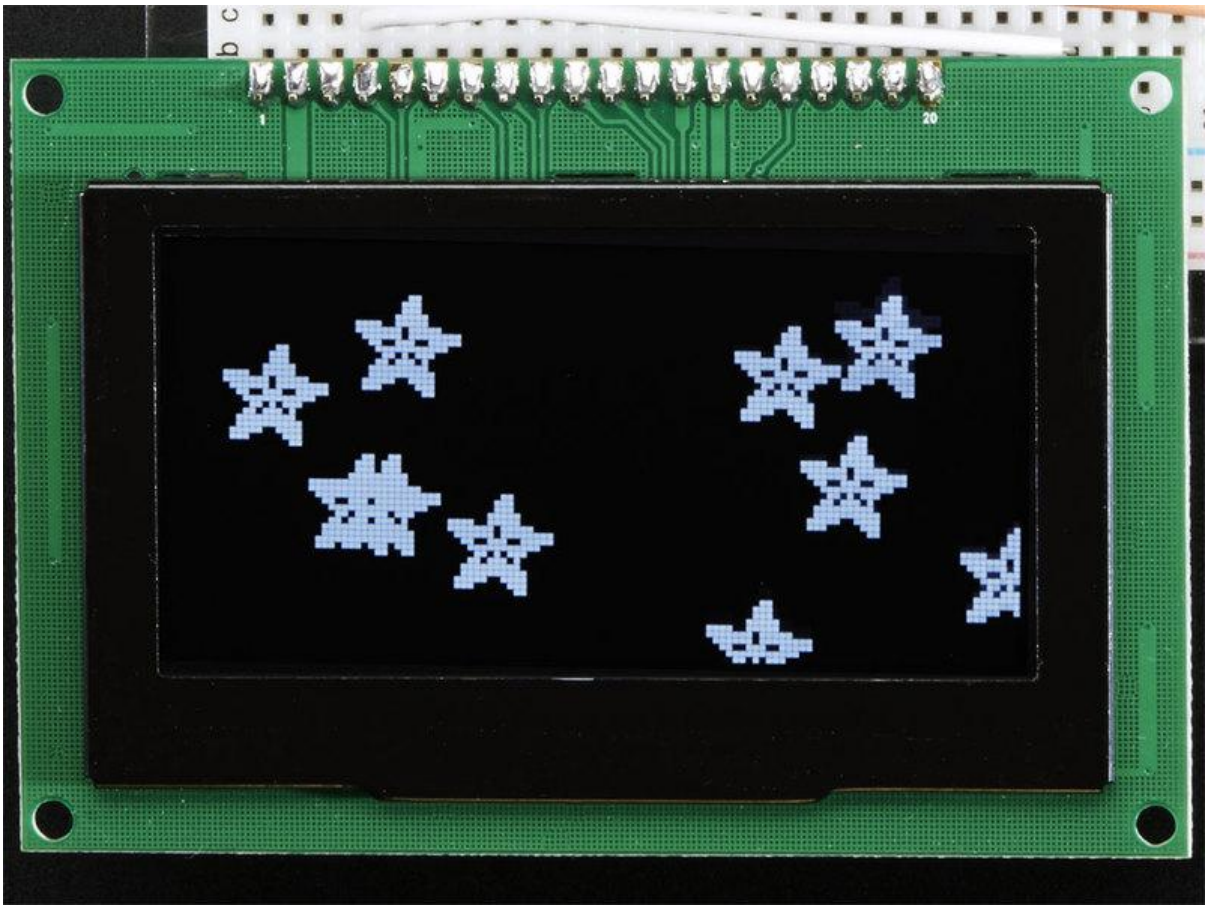
2.7" Monochrome 128x64 OLED Display Module

Created by lady ada

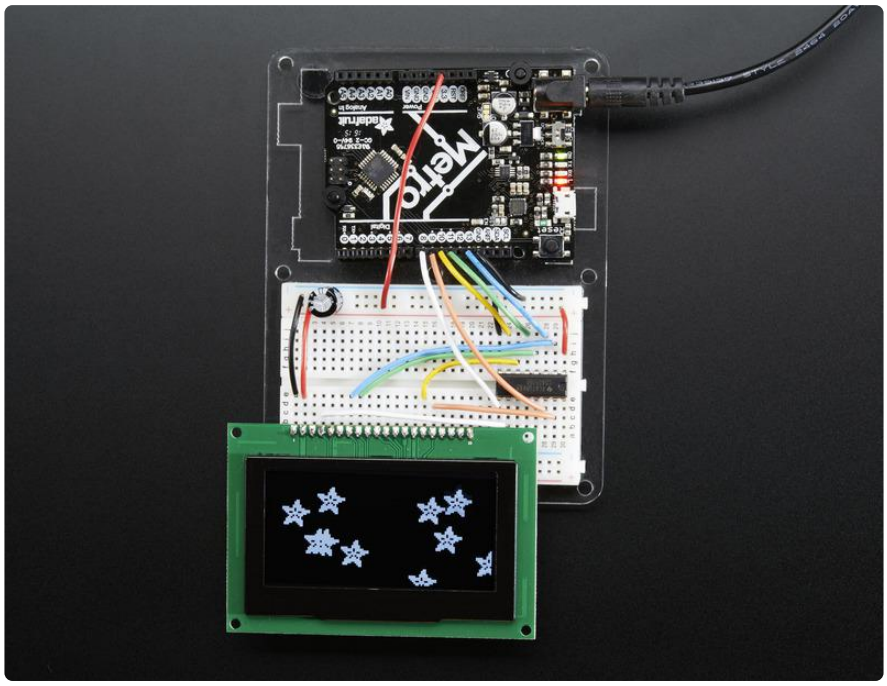


https://learn.adafruit.com/2-7-monochrome-128x64-oled-display-module

Last updated on 2023-08-29 02:59:04 PM EDT

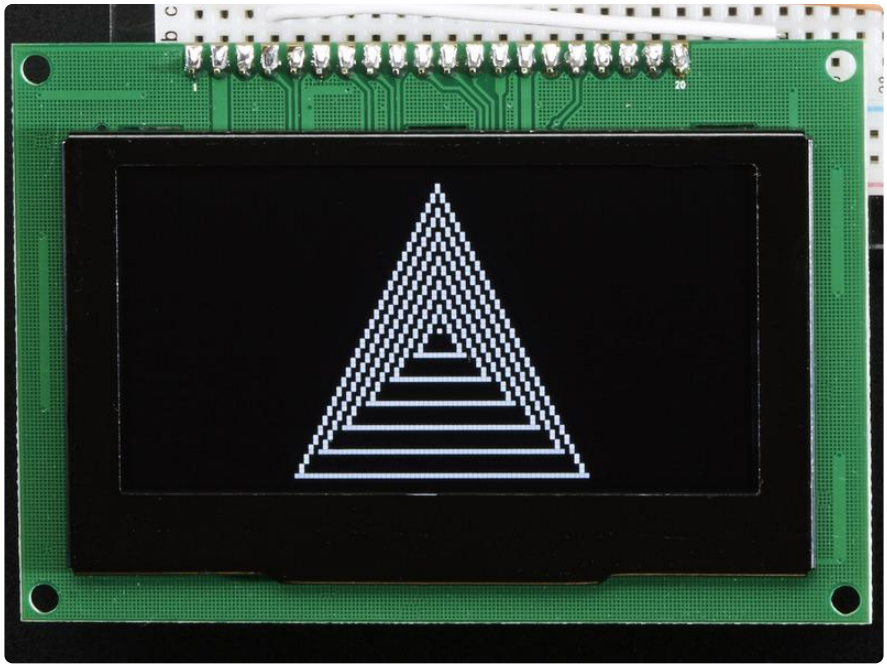
Table of Contents

Overview

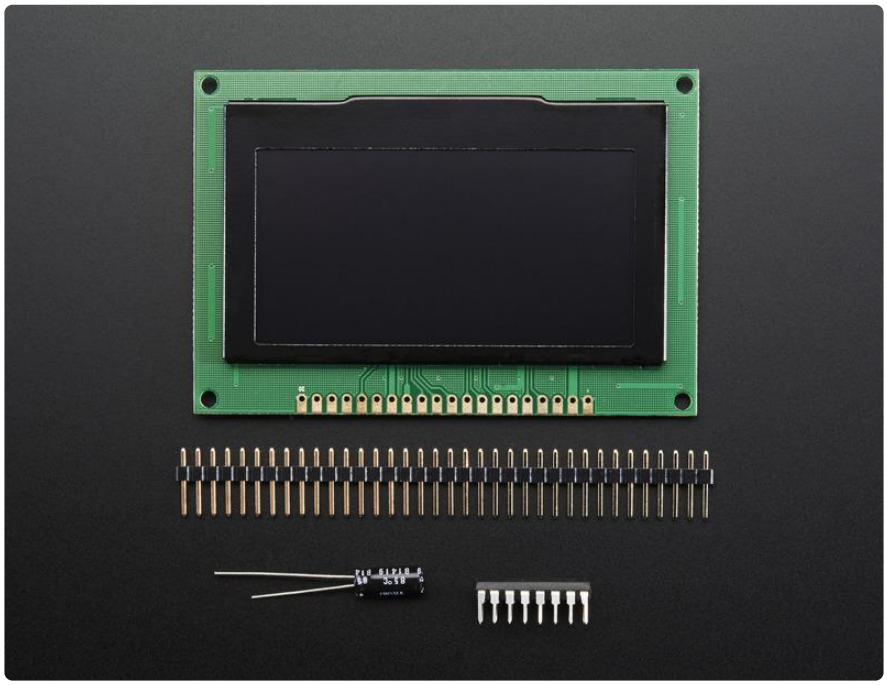


If you've been diggin' our monochome OLEDs but need something bigger, this display will delight you. These displays are 2.7" diagonal, and very readable due to the high contrast of an OLED display. This display is made of 128x64 individual white OLED pixels, each one is turned on or off by the controller 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!

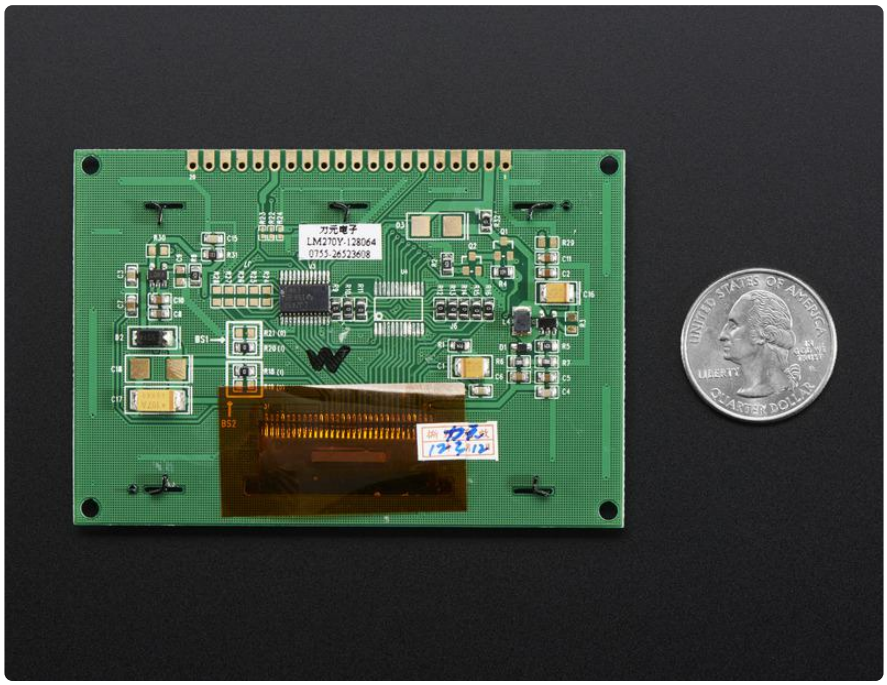
The driver chip, SSD1325 can communicate in two ways: 8-bit or SPI. Personally we think SPI is the way to go, only 4 or 5 wires are required. The OLED itself requires a 3.3V power supply and 3.3V logic levels for communication. We include a breadboard-friendly level shifter that can convert 3V or 5V 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 50-150mA 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.

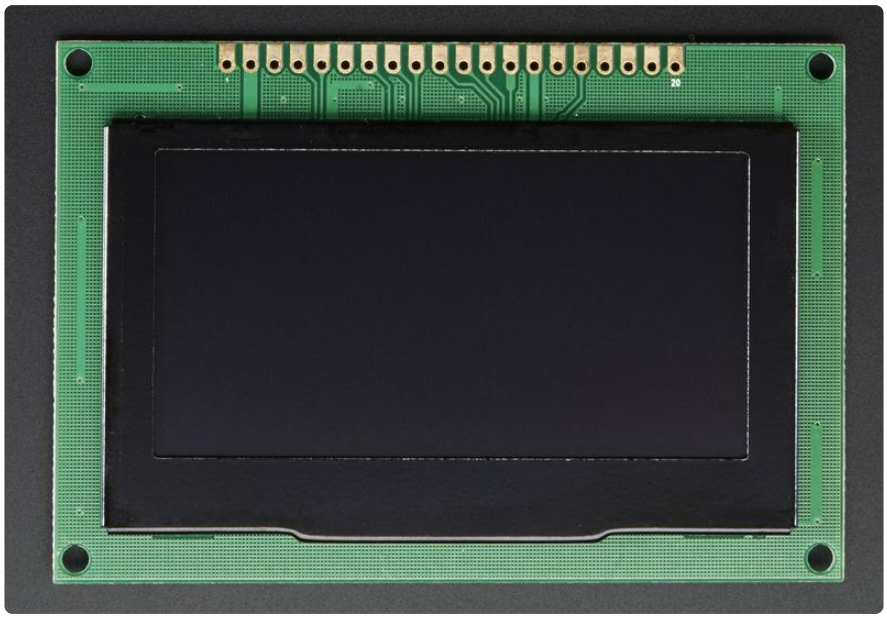


Each order comes with one assembled OLED module with a nice bezel and 4 mounting holes. The display is 3V logic & power so we include a HC4050 level shifter. We also toss in a 220uF capacitor, as we noticed an Arduino may need a little more capacitance on the 3.3V power supply for this big display! This 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 1K of RAM since the display must be buffered. The library can print text, bitmaps, pixels, rectangles, circles and lines. It uses 1K 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 100-150mA 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
- 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
- Pin #7 is Data0 - this pin is the SPI Clock pin and the 8-bit data bit 0 pin. This is a 3V logic level input pin when used with SPI, and an input/output when used in 8-bit.
- Pin #8 is Data1 - this pin is the SPI Data In pin and the 8-bit data bit 1 pin. This is a 3V logic level input pin when used with 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
- 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 both 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.

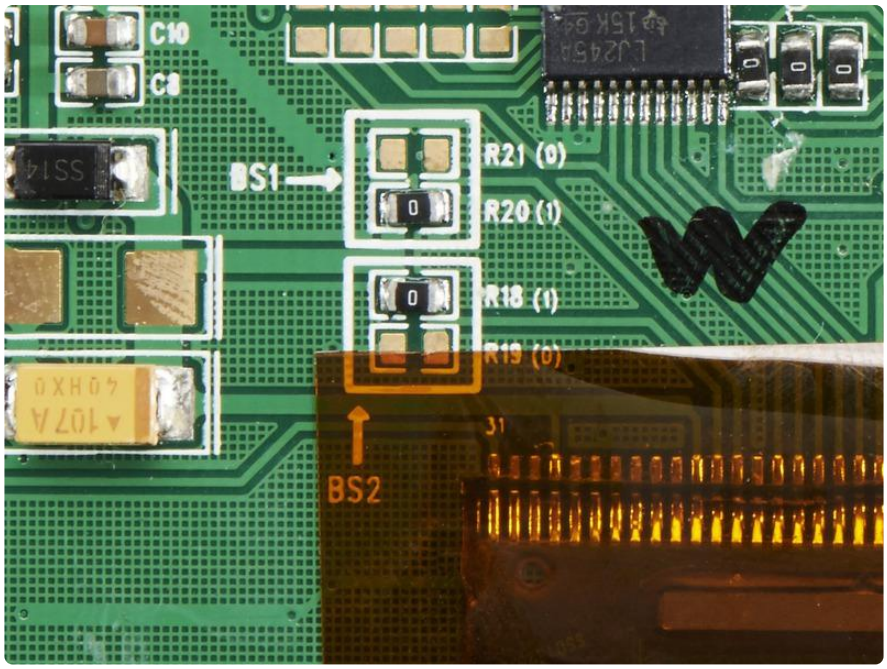
Assembly

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.

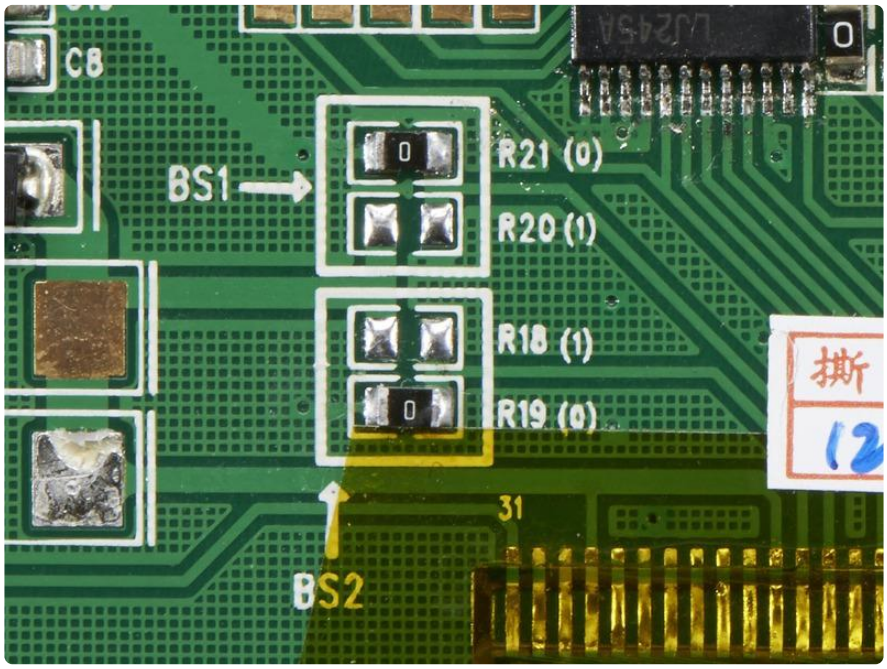
8-Bit "6800" mode

Your module probably came with this setting by default. The R20 and R18 resistors are in place and the R21 and R19 are missing.



SPI Mode

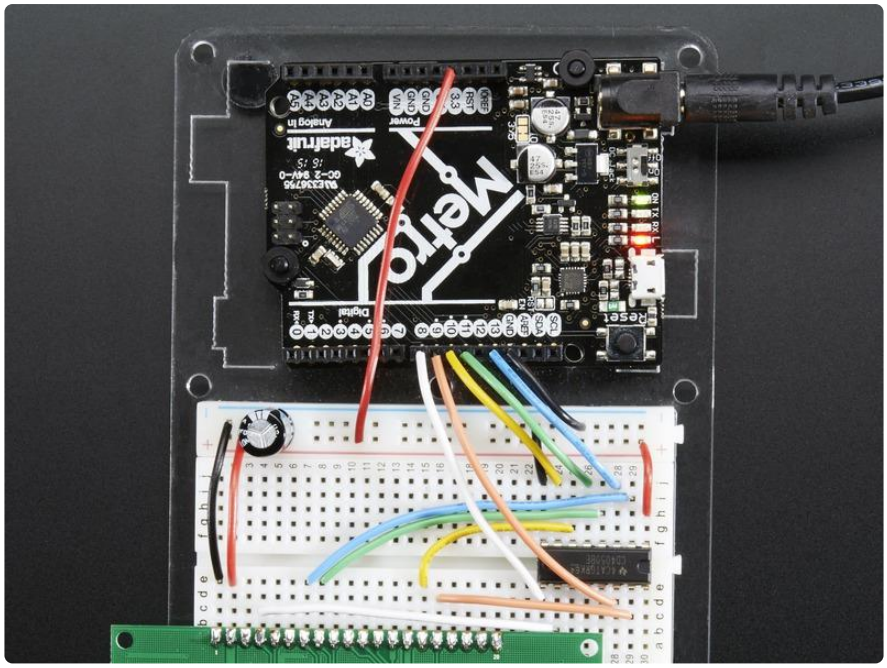
This is the mode you likely want to be in. You'll need to remove the R18 and R20 resistors by heating up the resistor with a soldering iron and maybe even melting a little solder on. Then remove the resistor and solder in R19 and R21 either by placing the resistors there (or, really, any 0-10K 0805 resistor) or a piece of wire.



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



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

SPI Wiring

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)

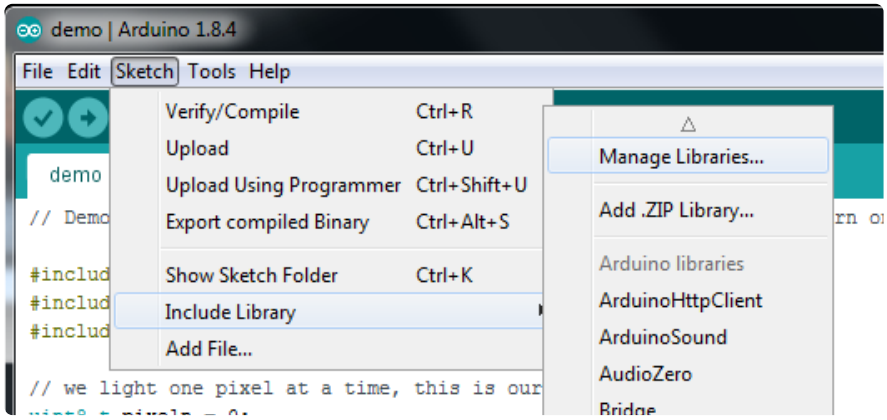
3.3V Capacitor

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

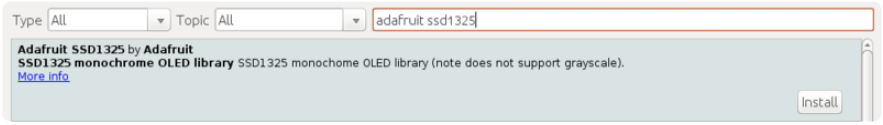
Download Libraries

To begin reading sensor data, you will need to download Adafruit_SSD1325 () and Adafruit_GFX () from the Arduino library manager.

Open up the Arduino library manager:



Search for the Adafruit SSD1325 library and install it:



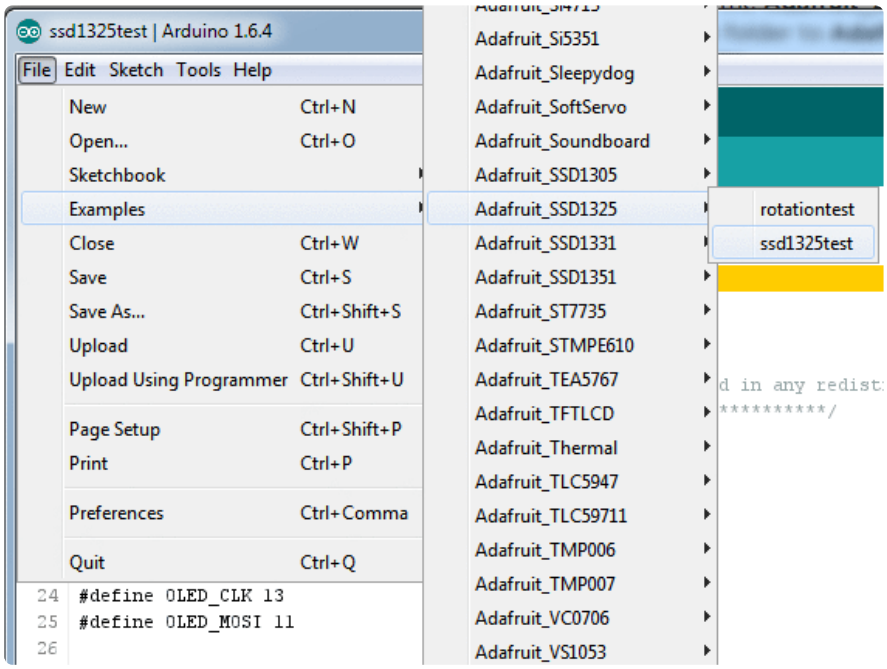
Search for the Adafruit GFX library and install it:

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

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

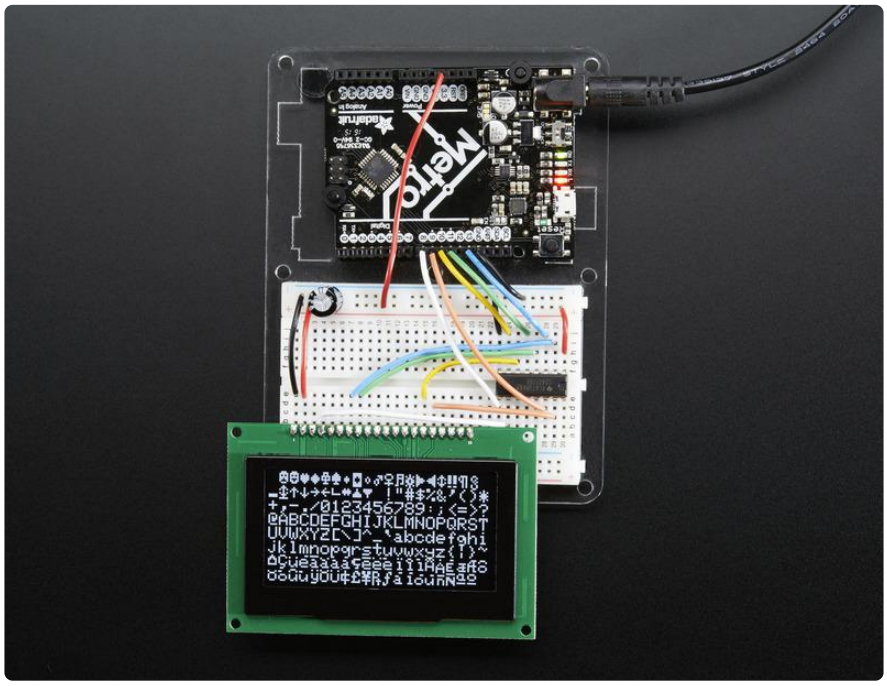
Running the Demo

After restarting the Arduino software, you should see a new example folder called Adafruit_SSD1325 and inside, an example called ssd1325test



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



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;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 ())

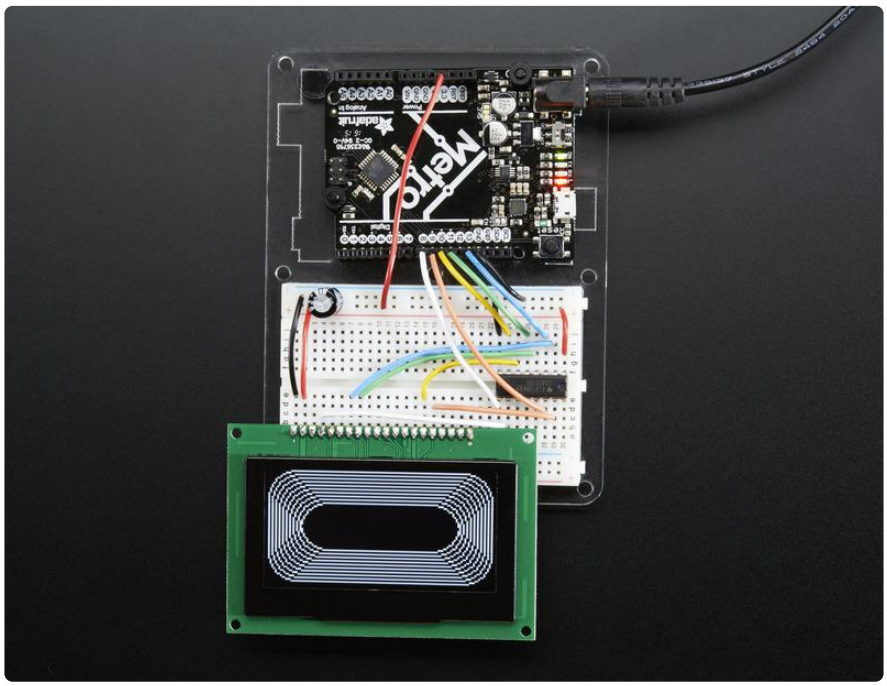
To enable hardware SPI, look for these lines:

```
// this is software SPI, slower but any pins
Adafruit_SSD1325 display(OLED_MOSI, OLED_CLK, OLED_DC, OLED_RESET, OLED_CS);

// this is for hardware SPI, fast! but fixed oubs
//Adafruit_SSD1325 display(OLED_DC, OLED_RESET, OLED_CS);
```

Comment out the top line and uncomment the bottom line

Using Adafruit GFX



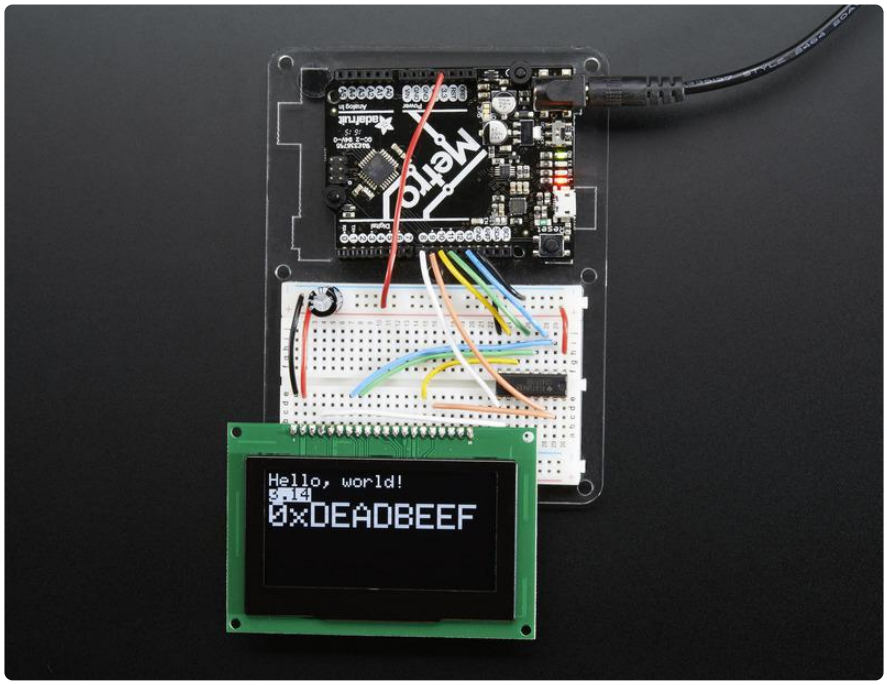
The Adafruit_GFX library for Arduino provides a common syntax and set of graphics functions for all of our TFT, LCD and OLED displays. This allows Arduino sketches to

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

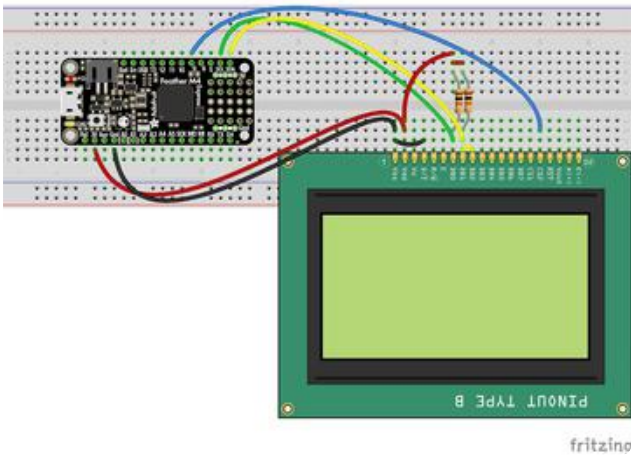
You can use this sensor with any CircuitPython microcontroller board or with a computer that has GPIO and Python thanks to Adafruit_Blinka, our CircuitPython-for-Python compatibility library ().

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.

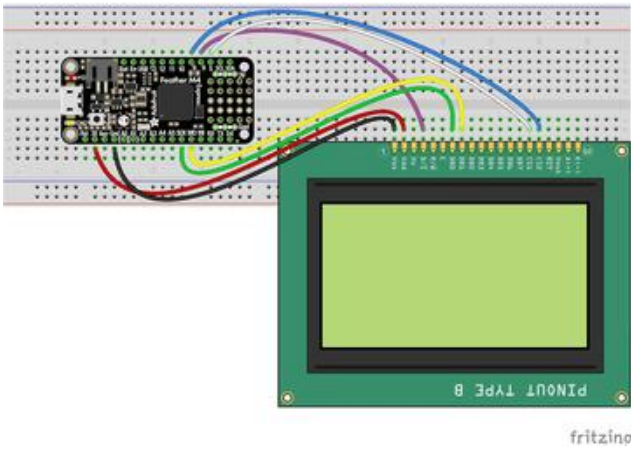
There's no SSD1305 or SSD1325 Large OLED Fritzing objects, so we sub'd a Graphic LCD in

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 SSD1325 Library

To use the SSD1325 OLED with your Adafruit CircuitPython board you'll need to install the Adafruit CircuitPython DisplayIO SSD1325 () 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_ssd1325

Before continuing make sure your board's lib folder or root filesystem has the adafruit_displayio_ssd1325.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 the the Adafruit CircuitPython Display Text library so the code didn't get overly complicated. Go ahead and install that 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 SSD1325 () 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_ssd1325
display_bus = displayio.I2CDisplay(i2c, device_address=0x3c)
```

Finally, you can pass the display_bus in and create an instance of the SSD1325 I2C driver by running:

```
display = adafruit_ssd1325.SSD1325(display_bus, width=128, height=64)
```

Now you should be seeing an image of the REPL. Note that the last two parameters to the `SSD1325` 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)
display = adafruit_ssd1325.SSD1325(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_ssd1325

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_ssd1325.SSD1325(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 `SSD1325` 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_ssd1325

displayio.release_displays()
```

```
# 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=board.D9
)

# 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)

WIDTH = 128
HEIGHT = 64
BORDER = 8
FONTSCALE = 1

display = adafruit_ssd1325.SSD1325(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=0x888888)
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_ssd1325` driver.

```
import board
import displayio
```

```
import terminalio
from adafruit_display_text import label
import adafruit_ssd1325
```

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()
```

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 SSD1325 needs to be slowed down to 1MHz, so we pass in the additional `baudrate` parameter. We also pass `board.D9` 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=board.D9)
```

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)
```

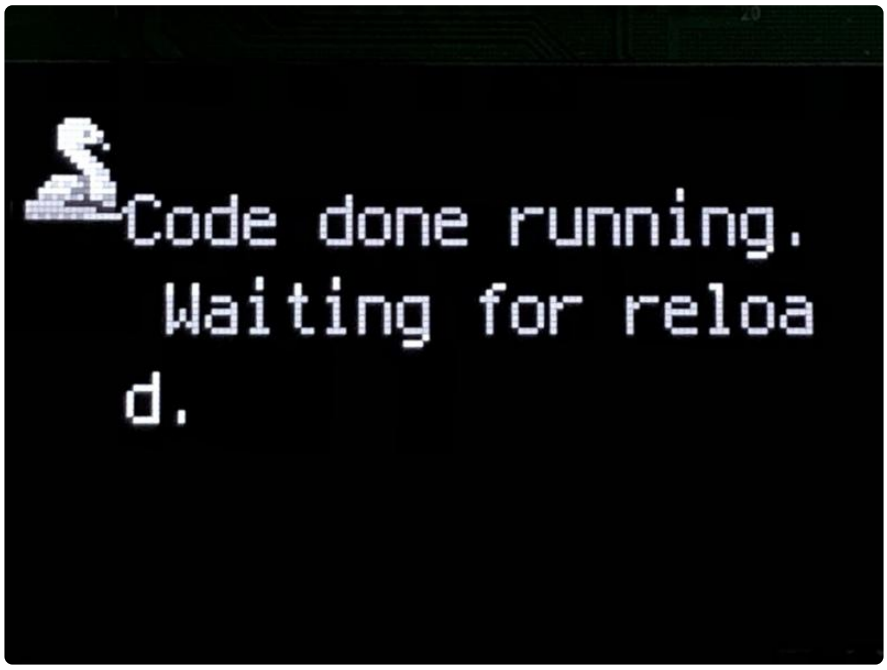
In order to make it easy to change display sizes, we'll define a few variables in one spot here. We have `WIDTH`, which is the display width, `HEIGHT`, which is the display height and `BORDER`, which we will explain a little further below. `FONTSCALE` will be the multiplier for the font size. If your 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. You may notice Blinka is grayscale.

```
display = adafruit_ssd1325.SSD1325(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.

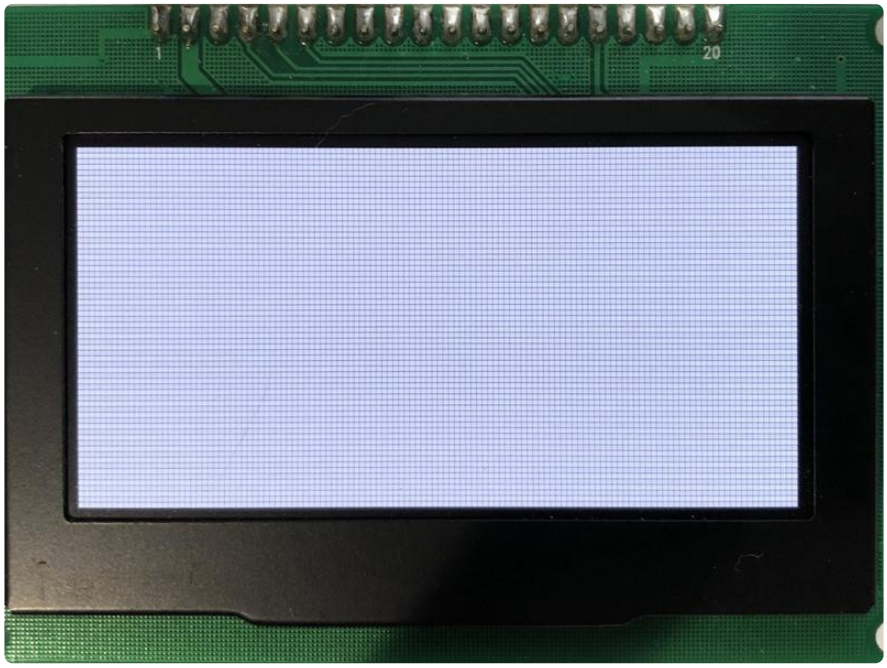
```
splash = displayio.Group(max_size=10)
display.show(splash)
```

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, 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 would end up some shade of gray.

```
color_bitmap = displayio.Bitmap(WIDTH, HEIGHT, 1)
color_palette = displayio.Palette(1)
color_palette[0] = 0x888888
```

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)
```

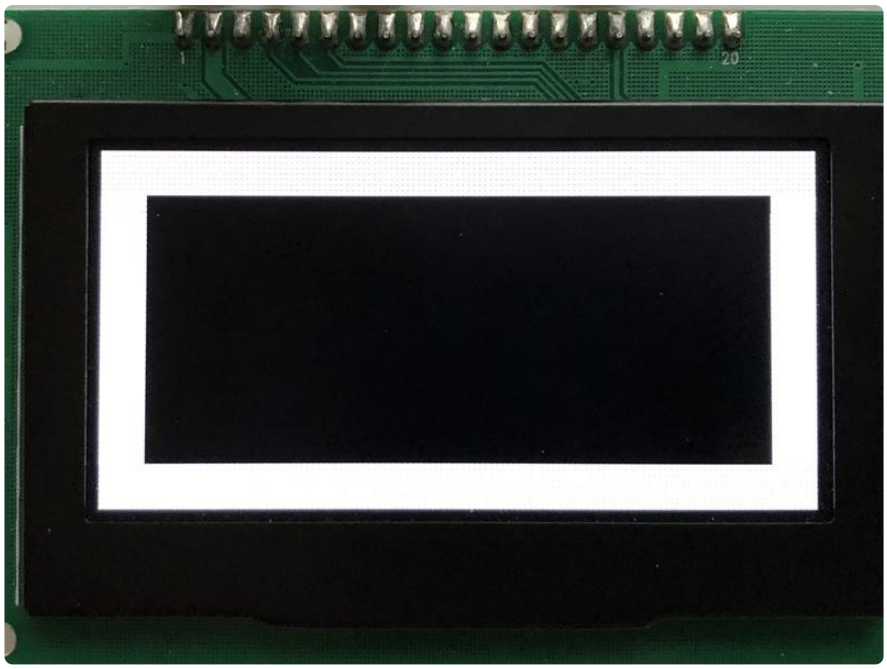


Next we will create a smaller black rectangle. The easiest way to do this is to create a new bitmap that is a little smaller than the full screen with a single color of `0x000000`, which is black, and place it in a specific location. In this case, we will create a bitmap that is 5 pixels smaller on each 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. Because of the way the OLED scans, you may notice the colors aren't distributed evenly.



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 gray text, so we'll pass it a value of `0x888888`. This display handles grayscale and this color is about halfway between `0x000000` and `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
```

Where to go from here

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

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_SSD1325.cpp in the libraries/Adafruit_SSD1325 folder and find these lines

---

```
static uint8_t buffer[SSD1325_LCDHEIGHT * SSD1325_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[SSD1325_LCDHEIGHT * SSD1325_LCDWIDTH / 8] = {`
and before `};`

Downloads

Datasheets:

- SSD1325 OLED driver () datasheet, this is the chip in the module that converts SPI/8-bit commands to OLED control signals
- Specifications for the UG-2864ASWIG01 OLED display screen in the module ()
- We also have a datasheet for the yellow module version of this display which is probably helpful and has almost identical specifications ()
- Mechanical specifications for the Yellow version of the OLED display itself ()