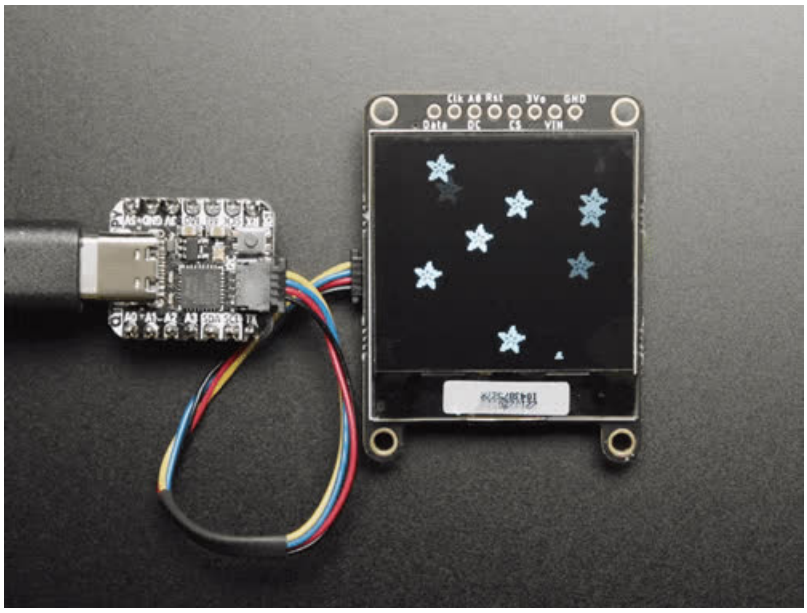
Adafruit Grayscale 1.5" 128x128 OLED Display

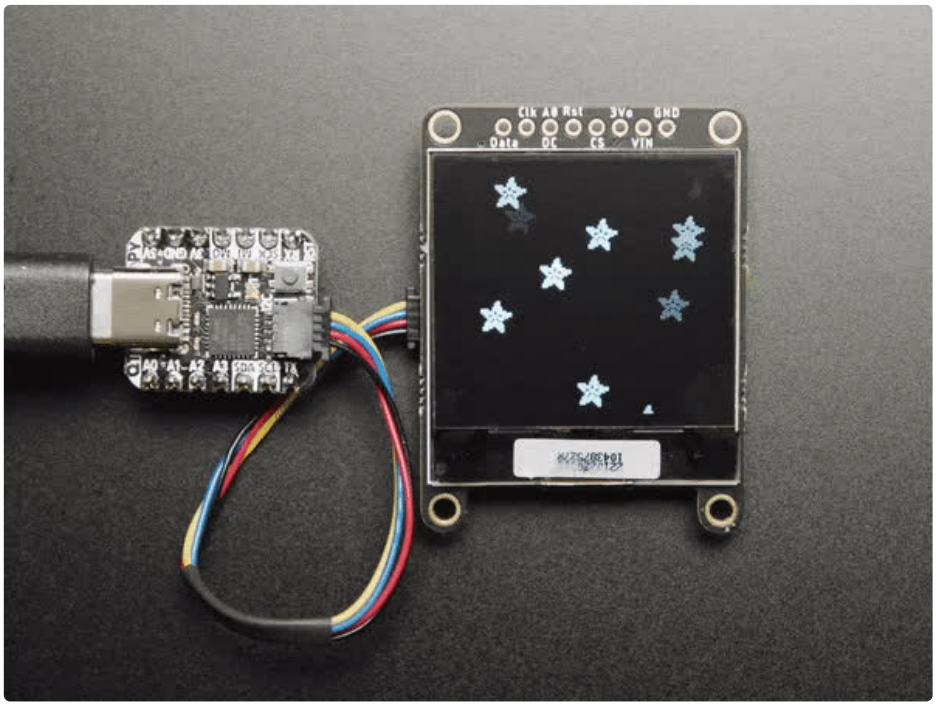
Created by Melissa LeBlanc-Williams



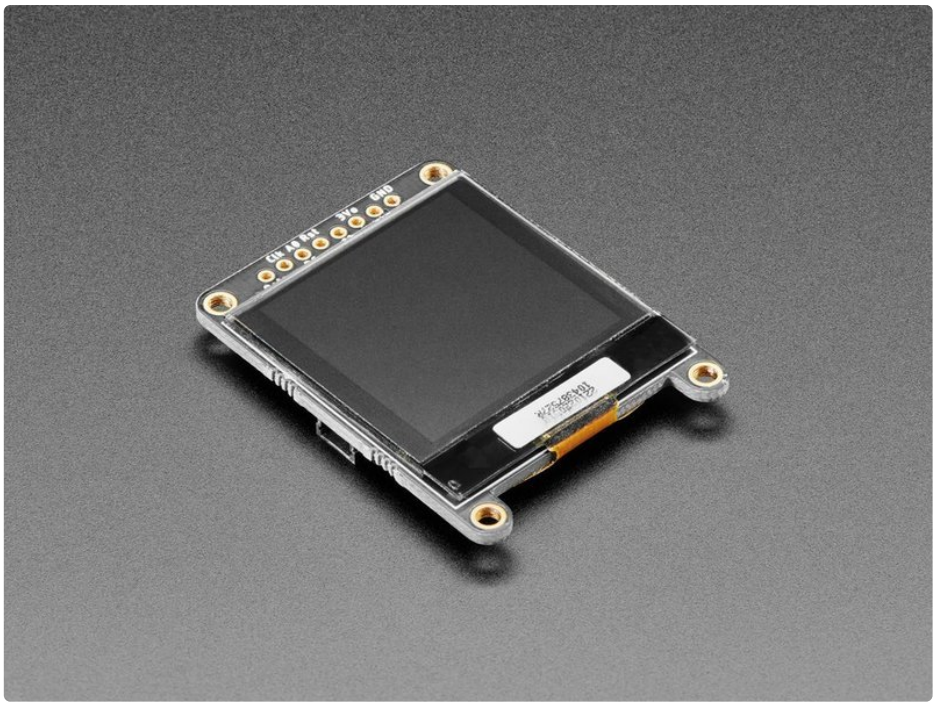
Last updated on 2021-09-07 12:35:02 PM EDT

Guide Contents

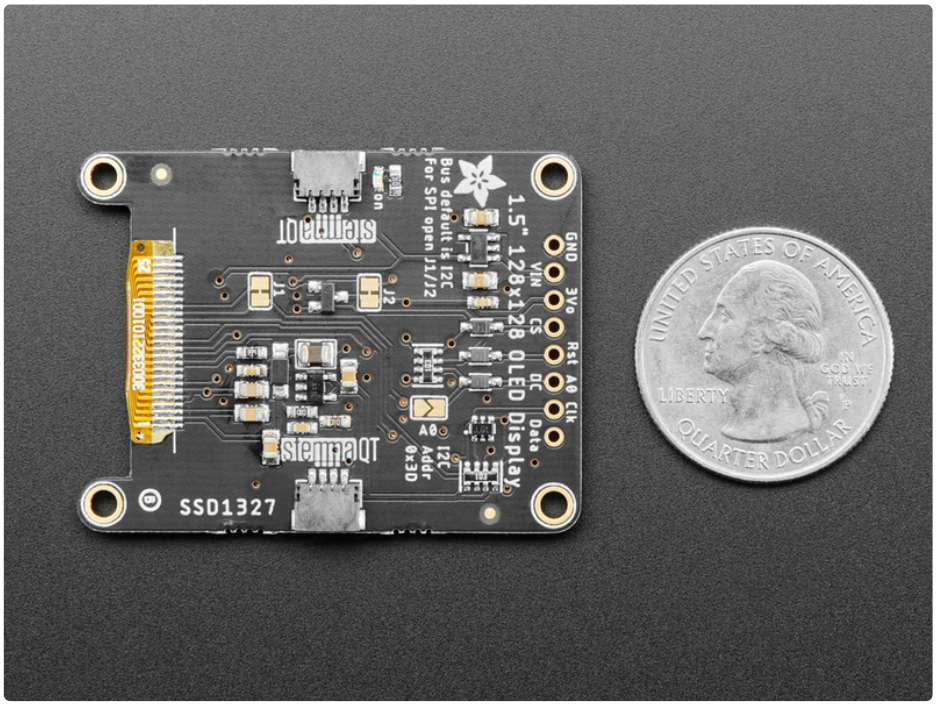
Overview



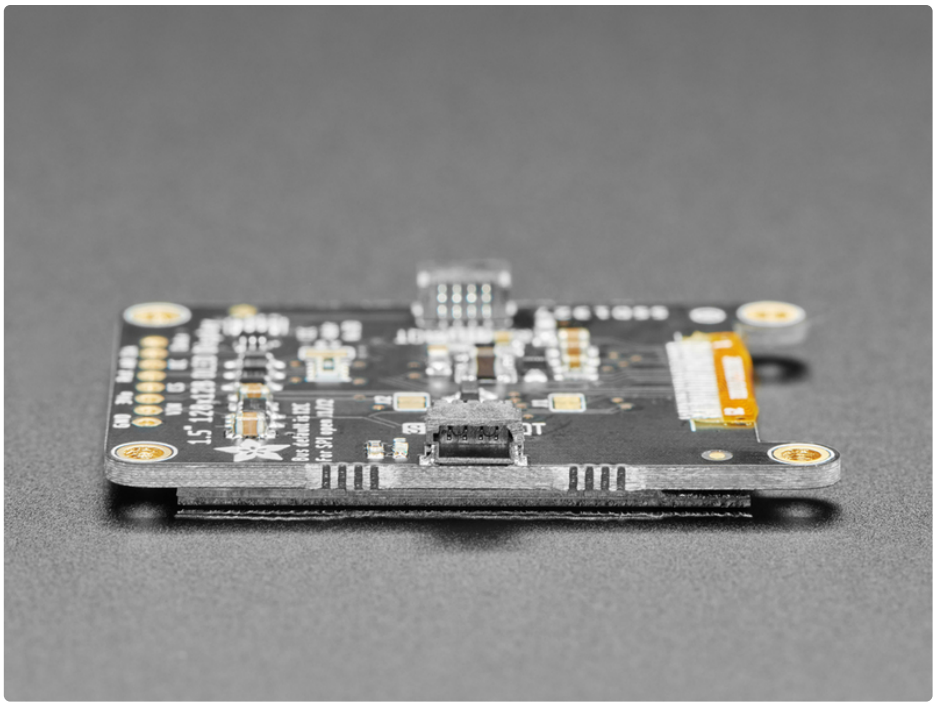
This OLED goes out to all the fans who want *more pixels*! Normally our 128x64 OLEDs are the biggest ones we've stocked that can use I2C. This one is a whopping 128x128 pixels and it even has an extra bonus - it can do grayscale pixels! Yep, you get the same crispness of a monochrome OLED but with 16 levels of gray.



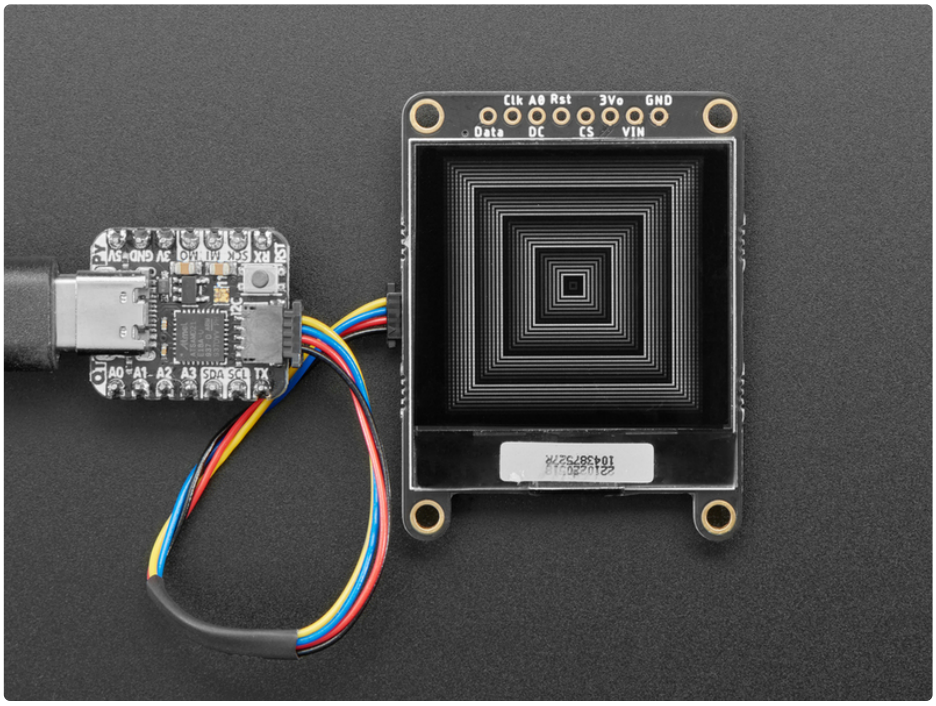
This display is a petite 1.5" diagonal, but very readable due to the high contrast of an OLED display. This display is made of 128x128 individual grayscale OLED pixels, each one is turned on or off by the controller chip. Because the display makes its own light, no backlight is required. This reduces the power required to run the OLED and is why the display has such high contrast; we really like these miniature displays for their crispness!



The SSD1327 driver chip can communicate in two ways: I2C or SPI. The OLED itself require a 3.3V and 12V power supply and 3.3V logic levels for communication. We include a 3.3V regulator and 12V boost converter, and all pins are fully level shifted so you can use with 3V or 5V devices!



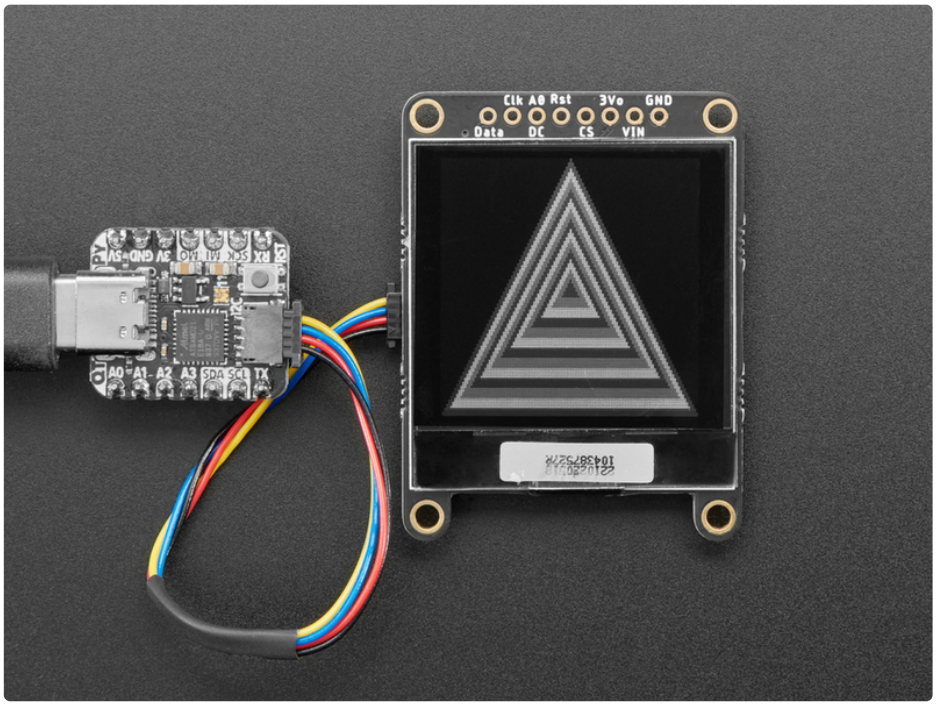
If you are using I2C, we've included SparkFun qwiic (https://adafru.it/Fpw) compatible STEMMA QT **(https://adafru.it/Ft4)** connectors for the I2C bus so **you don't even need to solder!** Plug and play with any board that has a Qwiic or STEMMA QT connector for effortless prototyping and development.



This display, being 16-level (4-bit) grayscale and 128x128 requires 128 * 128 * 4 bits = 8KB of SRAM to buffer. So **you can't use it with a small chip such as the Arduino UNO** (ATmega328 or 32u4). Pick a microcontroller or microcomputer with 16KB+ RAM - a SAMD21, SAMD51, ESP, nRF52, Teensy, etc will do an excellent job. As long as you have an I2C or SPI interface available, you're good to go - SPI will be

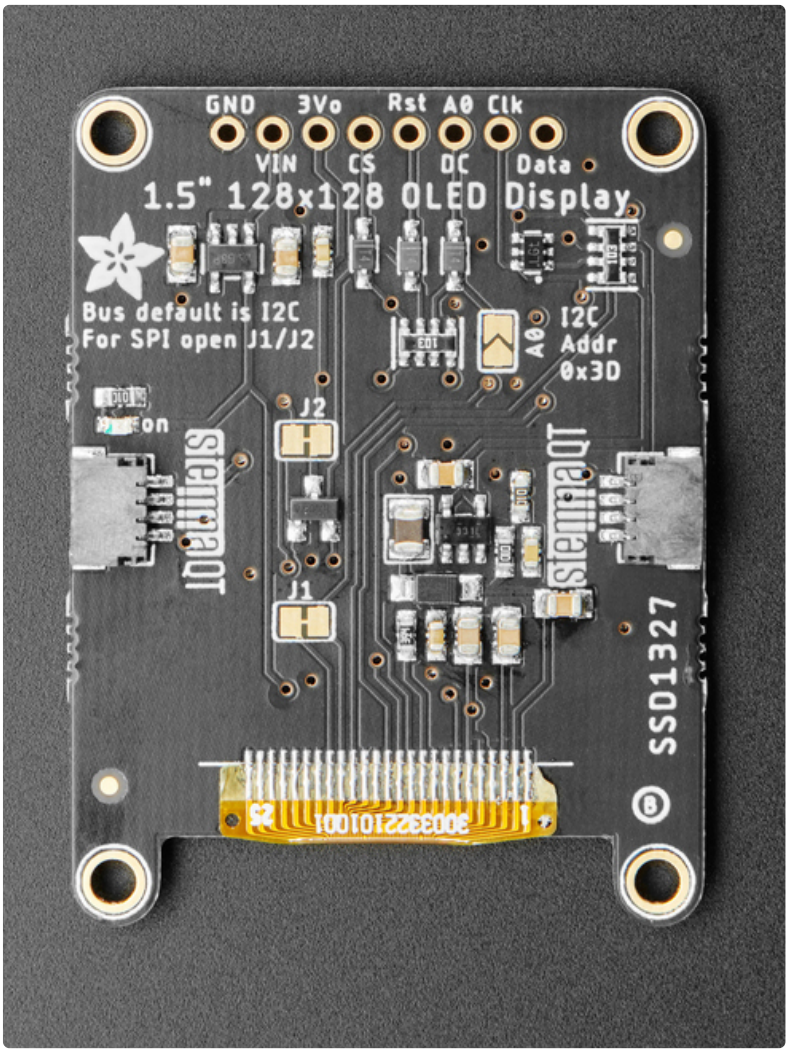
much faster but I2C requires fewer pins.

We have both Arduino (https://adafru.it/OvD) and CircuitPython support (https://adafru.it/OvE) for this display.

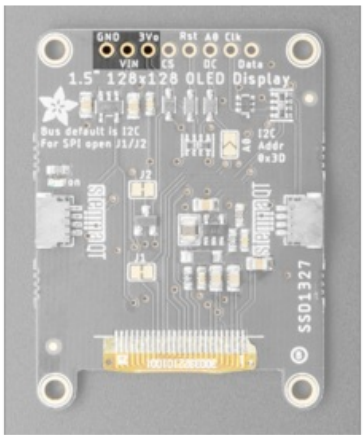


Please note that OLED displays are made of hundreds of...OLEDs! That means each pixel is a little organic LED, and if it's kept on for over 1000 hours, it'll start to dim. If you want to keep the display uniformly bright, please turn off the display (set the pixels off) when it isn't needed to keep them from dimming.

Pinouts

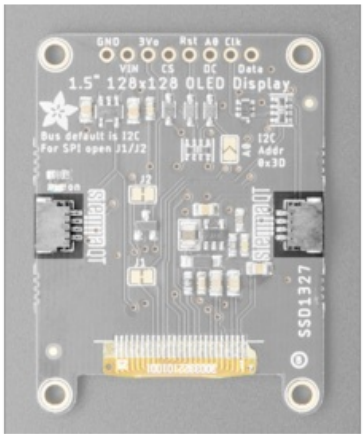


Power Pins



- **GND** - this is the power and signal ground pin
- **Vin** - this is the power pin, connect to 3-5VDC - it has reverse polarity protection but try to wire it right!
- **3Vo** - this is the 3.3V output from the onboard regulator, you can 'borrow' about 100mA if you need to power some other 3.3V logic devices

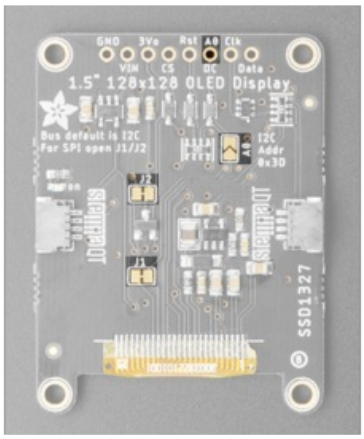
StemmaQT Ports



The display comes with 2 StemmaQT ports. Normally you would use one for an input and one for an output in order to chain multiple StemmaQT devices. Since they are connected in parallel, either side can be used as an input or output.

I2C and SPI Jumpers

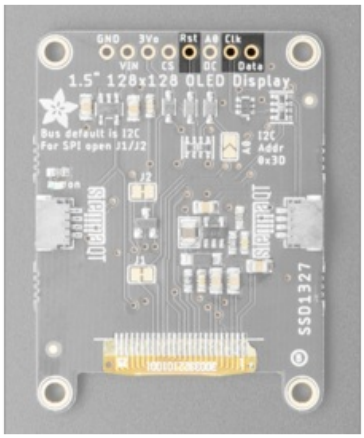
The Stemma QT ports will only work when the display is in I2C Mode.



You'll notice there are a few solderable jumpers on the back. To use in I2C mode, you'll want to make sure the **J1 and J2** are bridged. To use in SPI mode, you'll want to cut the traces between the jumpers for both J1 and J2. The board comes in I2C mode by default.

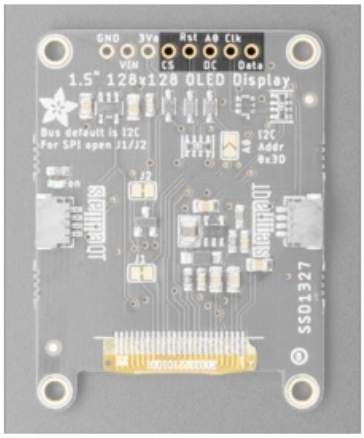
The **A0 jumper** is used for setting the I2C address. When it is open, the I2C address is `0x3D`. When it is bridged, the I2C Address is `0x3C`. Alternatively, by tying the A0 pin to ground, you can set the I2C address to `0x3C`.

I2C Pins



- **Rst** - this is the Reset pin. You may be able to share this with your microcontroller reset pin but, if you can, connect it to a digital pin.
- **Clk** - this is the Clock pin and is connected to **I2C SCL**
- **Data** - this is the Data pin and is connected to **I2C SDA**

SPI Pins

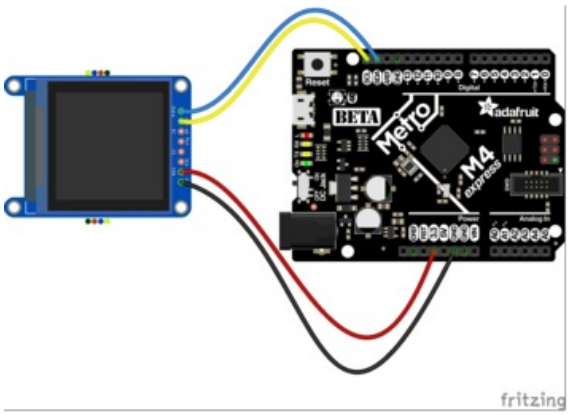


- **CS** - this is the Chip Select pin, it is used by SPI to select the device.
- **Rst** - this is the Reset pin, you may be able to share this with your microcontroller reset pin but if you can, connect it to a digital pin.
- **DC** - this is the Data/Command selector pin used to differentiate between Data and Commands that are sent to the display.
- **Clk** - this is the Clock pin and is connected to **SPI SCLK**
- **Data** - this is the Data pin and is connected to **SPI MOSI**

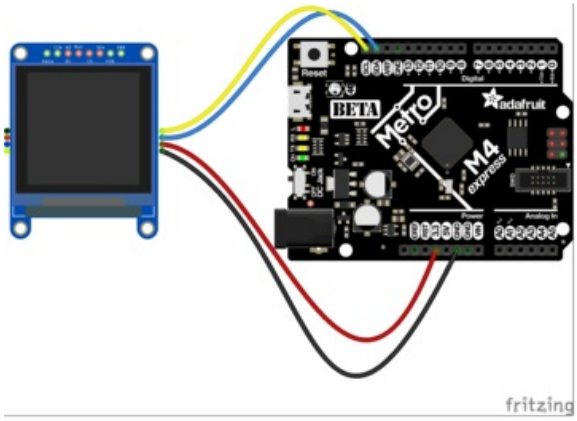
Arduino Wiring and Test

You can easily wire this breakout to any microcontroller - below, an Adafruit Metro is used. For another kind of microcontroller, as long as you have 4 available pins, it is possible to 'bit-bang SPI' or you can use two I2C pins, but usually those pins are fixed in hardware. Just check out the library, then port the code.

I2C Wiring



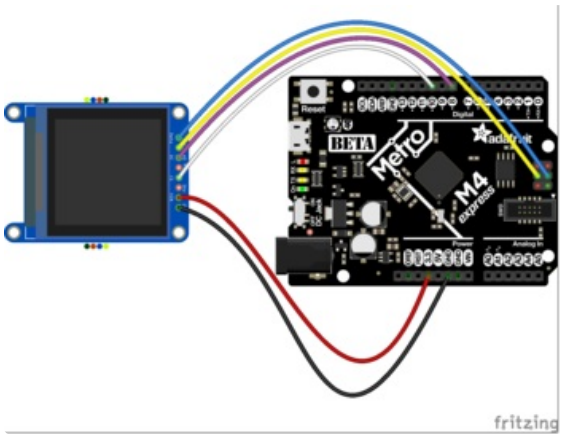
Use this wiring if you want to connect via I2C interface



- Connect **Vin (red wire on STEMMA QT version)** to the power supply, 3-5V is fine. Use the same voltage that the microcontroller logic is based off of. For most Arduinos, that is 5V
- Connect **GND (black wire on STEMMA QT version)** to common power/data ground
- Connect the **Clk (yellow wire on STEMMA QT version)** pin to the I2C clock **SCL** pin on your Arduino. On an UNO & '328 based Arduino, this is also known as **A5**, on a Mega it is also known as **digital 21** and on a Leonardo/Micro, **digital 3**
- Connect the **Data (blue wire on STEMMA QT version)** pin to the I2C data **SDA** pin on your Arduino. On an UNO & '328 based Arduino, this is also known as **A4**, on a Mega it is also known as **digital 20** and on a Leonardo/Micro, **digital 2**

SPI Wiring

If you want to use the board in SPI. mode, you need to cut J1 and J2 first. See Pinouts for more details.



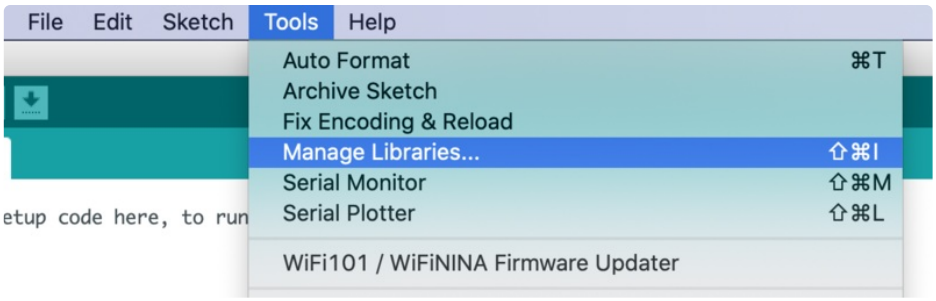
Since this is a SPI-capable display, we can use hardware or 'software' SPI. To make wiring identical on all Arduinos, we'll begin with 'software' SPI. The following pins should be used:

- Connect **Vin** to the power supply, 3-5V is fine. Use the same voltage that the microcontroller logic is based off of. For most Arduinos, that is 5V
- Connect **GND** to common power/data ground
- Connect the **Clk** pin to the SPI clock **SCK** pin on your Arduino. On an UNO & '328 based Arduino, this is also known as **13**. On boards such as the Metro M0/M4, you'll want to use the **ICSP Header Pin 3**.
- Connect the **Data** pin to the SPI data **MOSI** pin on your Arduino. On an UNO & '328 based Arduino, this is also known as **11**. On boards such as the Metro M0/M4, you'll want to use the **ICSP Header Pin 4**.
- Connect the **DC** pin to **D8** on the Arduino and similar shaped boards. If you don't have this pin on your Microcontroller, feel free to use a different one and update the example.
- Connect the **CS** pin to **D10** on the Arduino and similar shaped boards. If you don't have this pin on your Microcontroller, feel free to use a different one and update the example.

Download Libraries

To begin reading sensor data, you will need to download Adafruit_SSD1327 (https://adafru.it/OBj) and Adafruit_GFX (https://adafru.it/aJa) from the Arduino library manager.

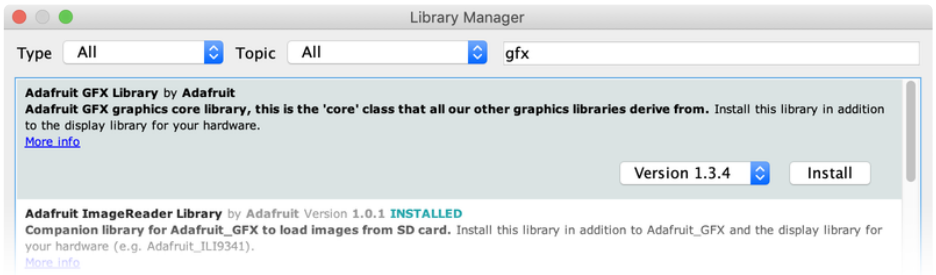
From the IDE open up the library manager...



nd type in **adafruit EPD** to locate the library. Click **Install**

If you would like to draw bitmaps, do the same with **adafruit SSD1327**, click **Install**

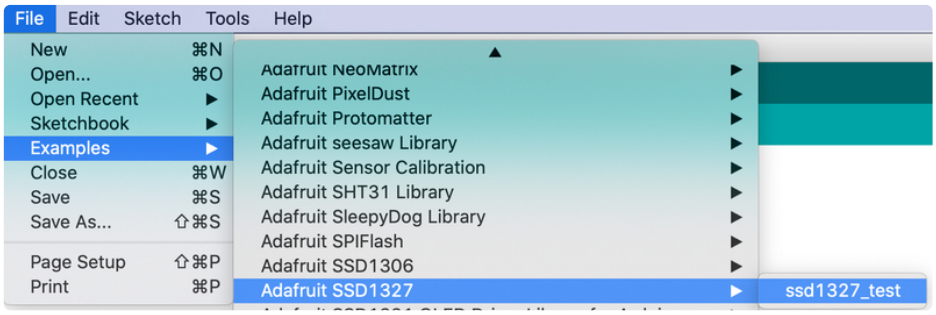
Do the same to install the latest **adafruit GFX** library, click **Install**



If using an earlier version of the Arduino IDE (pre-1.8.10), locate and install **Adafruit_BusIO** (newer versions handle this prerequisite automatically).

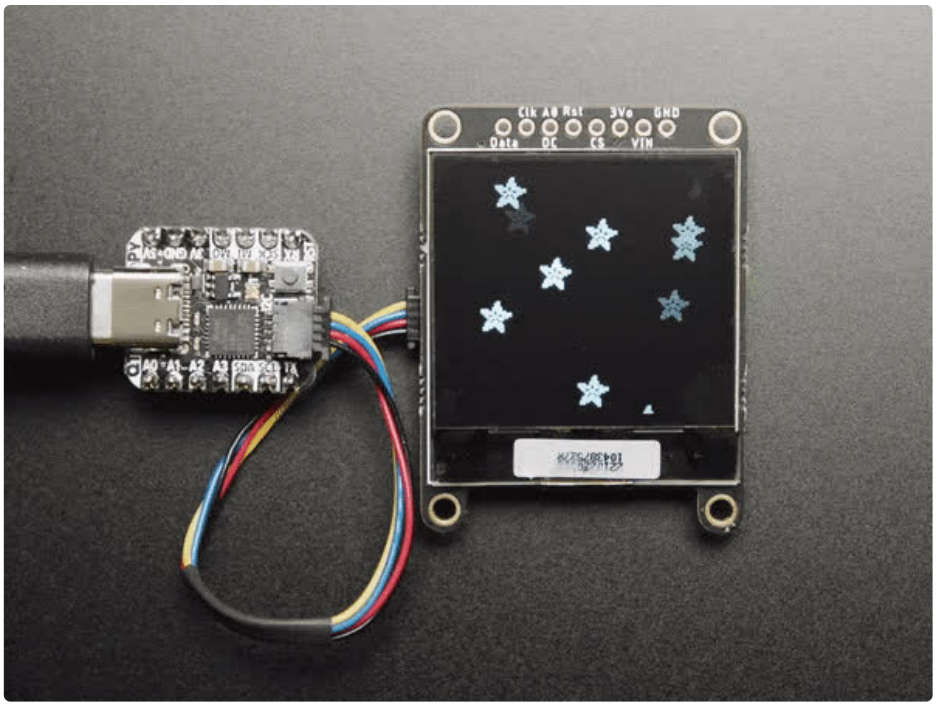
Run the Demo

Open up **File→Examples→Adafruit_SSD1327→ssd1327_test**



Make any necessary changes to the pins. If you are running in SPI mode, comment out the I2C Initializer and then either uncomment the hardware SPI or software SPI initializer depending on how you have it wired up and whether you want to use the hardware SPI pins or not.

Upload the sketch to your microcontroller and you should see a series of tests run concluding with some Adafruit flakes falling at the end.


Clk A0 Rst 3Vo GND
Data DC CS VIN

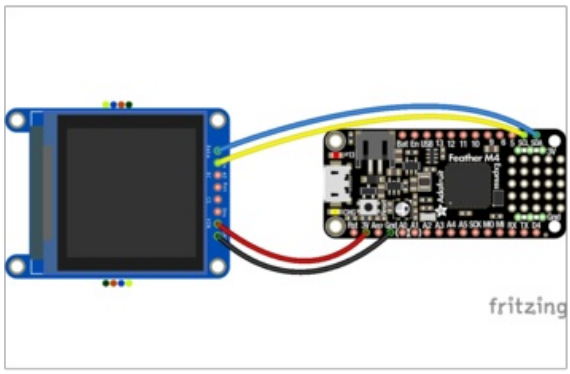
CircuitPython Wiring and Usage

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

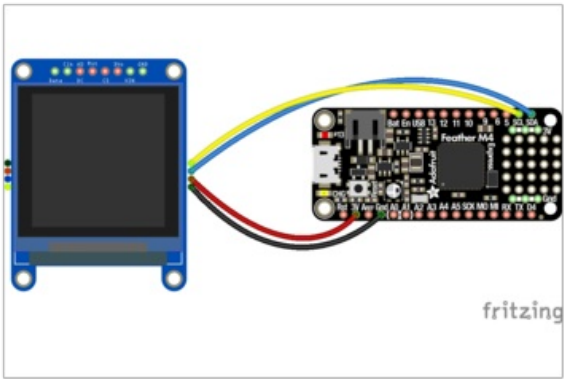
Connect the OLED to your microcontroller board as shown below.

I2C Wiring

Use this wiring if you want to connect via I2C interface

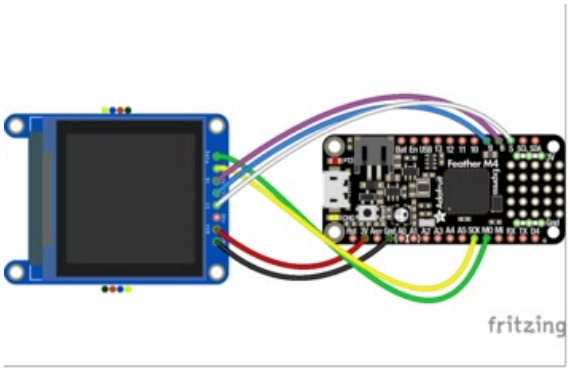


- Connect **Vin (red wire on STEMMA QT version)** to the power supply, 3-5V is fine. Use the same voltage that the microcontroller logic is based off of.
- Connect **GND (black wire on STEMMA QT version)** to common power/data ground
- Connect the **Clk (yellow wire on STEMMA QT version)** pin to the I2C clock **SCL** pin on your Microcontroller.
- Connect the **Data (blue wire on STEMMA QT version)** pin to the I2C data **SDA** pin on your Microcontroller.



SPI Wiring

Since this is a SPI-capable display, we can use SPI. The following pins should be used:



- Connect **Vin** to the power supply, 3-5V is fine. Use the same voltage that the microcontroller logic is based off of.
- Connect **GND** to common power/data ground
- Connect the **Clk** pin to the SPI clock **SCK** pin on your Microcontroller.
- Connect the **Data** pin to the SPI data **MOSI** pin on your Microcontroller.
- Connect the **CS** pin to **D5** on your Microcontroller. If you don't have this pin on your Microcontroller, feel free to use a different one and update the example.
- Connect the **DC** pin to **D6** on your Microcontroller. If you don't have this pin on your Microcontroller, feel free to use a different one and update the example.
- Connect the **Rst** pin to **D9** on your Microcontroller. If you don't have this pin on your Microcontroller, feel free to use a different one and update the example.

If you want to use the board in SPI. mode, you need to cut J1 and J2 first. See Pinouts for more details.

To use this grayscale OLED display with displayio, you will need to use the absolute latest version of CircuitPython and a board that can fit `displayio`. See the Support Matrix to determine if `displayio` is available on a given board: https://circuitpython.readthedocs.io/en/latest/shared-bindings/support_matrix.html

CircuitPython displayio Library Installation

To use displayio, you will need to install the adafruit_ssd1327 library for your display.

First make sure you are running the latest version of Adafruit CircuitPython (https://adafru.it/Amd) for your board. You will need the latest version of CircuitPython.

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 (https://adafru.it/zdx). Our introduction guide has a great page on how to install the library bundle (https://adafru.it/ABU) for both express and non-express boards.

You will need to copy the appropriate displayio driver from the bundle **lib** folder to a **lib** folder on your **CIRCUITPY** drive. The displayio driver contains the initialization codes specific to your display that are needed to for it to work.

When downloading CircuitPython, for Grayscale support you will need to choose Absolute Newest, choose your language, and then download the top-most link.

Adafruit_CircuitPython_SSD1327

This display uses the Adafruit_CircuitPython_SSD1327 library. Copy the `adafruit_ssd1327.mpy` file from the bundle to the **lib** folder on your **CIRCUITPY** drive.

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.

Displayio is only available on express boards (or other higher-memory boards, such as QT Py RP2040) due to the smaller memory size on non-express boards. For QT Py M0, for example, use an M0 Haxpress.

Usage

It's easy to use OLEDs with Python and the Adafruit CircuitPython SSD1327 (https://adafru.it/OvE) 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_ssd1327
display_bus = displayio.I2CDisplay(i2c, device_address=0x3D)
```

Finally, you can pass the `display_bus` in and create an instance of the SSD1327 I2C driver by running:

```
display = adafruit_ssd1327.SSD1327(display_bus, width=128, height=128)
```

Now you should be seeing an image of the REPL. Note that the last two parameters to the `SSD1327` class initializer are the **width** and **height** of the display in pixels.

Changing the I2C address

If you connect the **A0 Pin** of the OLED to **Ground** instead of **+3V** or create a solder bridge on the back of the display for the **A0 jumper** the I2C address will be different different (`0x3c`):

```
display_bus = displayio.I2CDisplay(i2c, device_address=0x3c)
display = adafruit_ssd1327.SSD1327(display_bus, width=128, height=128)
```

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_ssd1327

displayio.release_displays()

spi = board.SPI()
oled_cs = board.D5
oled_dc = board.D6
oled_reset = board.D9

display_bus = displayio.FourWire(spi, command=oled_dc, chip_select=oled_cs,
                                 reset=oled_reset, baudrate=1000000)
display = adafruit_ssd1327.SSD1327(display_bus, width=128, height=128)
```

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 `SSD1327` class initializer are the **width** and **height** of the display in pixels.

Example Code

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

import board
import displayio
import terminalio
from adafruit_display_text import label
import adafruit_ssd1327

displayio.release_displays()

# Use for I2C
i2c = board.I2C()
display_bus = displayio.I2CDisplay(i2c, device_address=0x3D)

# 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
# )

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

display = adafruit_ssd1327.SSD1327(display_bus, width=WIDTH, height=HEIGHT)

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

# Draw a background rectangle, but not the full display size
color_bitmap = displayio.Bitmap(
    display.width - BORDER * 2, display.height - BORDER * 2, 1
)
color_palette = displayio.Palette(1)
color_palette[0] = 0xFFFFFF  # White
bg_sprite = displayio.TileGrid(
    color_bitmap, pixel_shader=color_palette, x=BORDER, y=BORDER
)
```

```
splash.append(bg_sprite)

# Draw a smaller inner rectangle
inner_bitmap = displayio.Bitmap(
    display.width - BORDER * 4, display.height - BORDER * 4, 1
)
inner_palette = displayio.Palette(1)
inner_palette[0] = 0x888888  # Gray
inner_sprite = displayio.TileGrid(
    inner_bitmap, pixel_shader=inner_palette, x=BORDER * 2, y=BORDER * 2
)
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_ssd1327` driver.

```
import board
import displayio
import terminalio
from adafruit_display_text import label
import adafruit_ssd1327
```

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 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=0x3D)
```

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 SSD1327 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.

```
# 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
)
```

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.

```
WIDTH = 128
HEIGHT = 128
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_ssd1327.SSD1327(display_bus, width=WIDTH, height=HEIGHT)
```

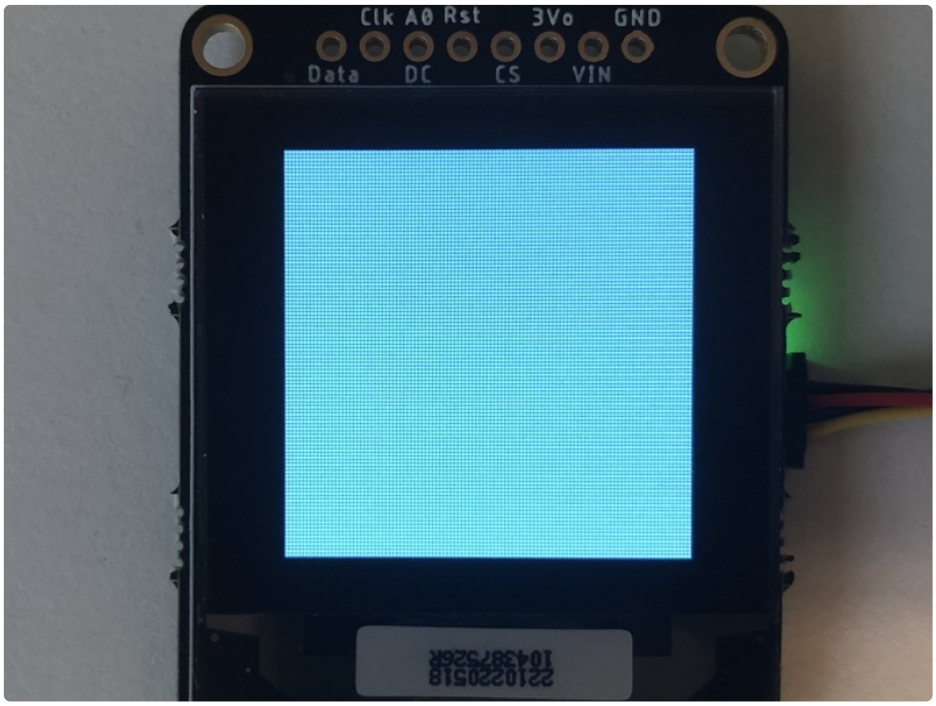
Next we create a background splash image. We do this by creating a group that we can add elements to and adding that group to the display. In this example, we are limiting the maximum number of elements to 10, but this can be increased if you would like. The display will automatically handle updating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

```
# Make the display context
splash = displayio.Group(max_size=10)
display.show(splash)
```

Next we create a Bitmap that is the full width and height of the display minus the value of the **BORDER** variable for each of the 2 sides. 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.

With all those pieces in place, we create a TileGrid by passing the bitmap and palette and draw it at `(8, 8)` so that it ends up centered.

```
# Draw a background rectangle, but not the full display size
color_bitmap = displayio.Bitmap(
    display.width - BORDER * 2, display.height - BORDER * 2, 1
)
color_palette = displayio.Palette(1)
color_palette[0] = 0xFFFFFF  # White
bg_sprite = displayio.TileGrid(
    color_bitmap, pixel_shader=color_palette, x=BORDER, y=BORDER
)
splash.append(bg_sprite)
```

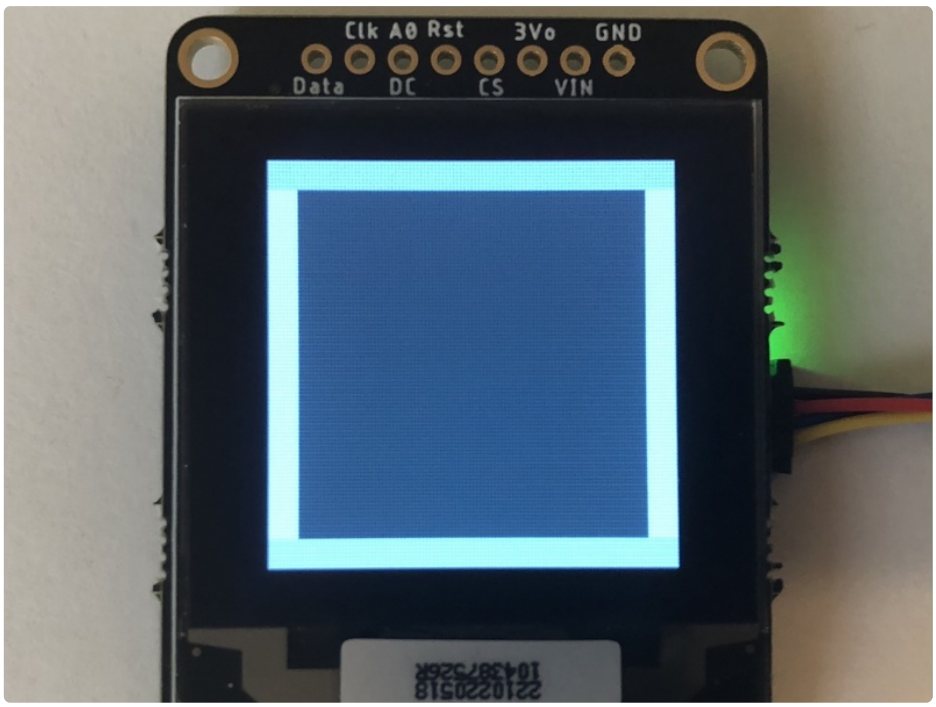


Next we will create a smaller gray rectangle. The easiest way to do this is to create a new bitmap that is a little smaller than the full screen with a single color of `0x888888`, which is **gray**, and place it in a specific location. This display handles grayscale and this color is about halfway between `0x000000` and `0xFFFFFF`. In this case, we will create a bitmap that is **8** pixels smaller on each side than the previous rectangle for a difference of 16 pixels on each side. We have the BORDER set to **8** , so we'll want to subtract 32 from each of those numbers.

We'll want to place it at the position `(16, 16)` so that it ends up centered as well.

```
# Draw a smaller inner rectangle
inner_bitmap = displayio.Bitmap(
    display.width - BORDER * 4, display.height - BORDER * 4, 1
)
inner_palette = displayio.Palette(1)
inner_palette[0] = 0x888888  # Gray
inner_sprite = displayio.TileGrid(
    inner_bitmap, pixel_shader=inner_palette, x=BORDER * 2, y=BORDER * 2
)
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.

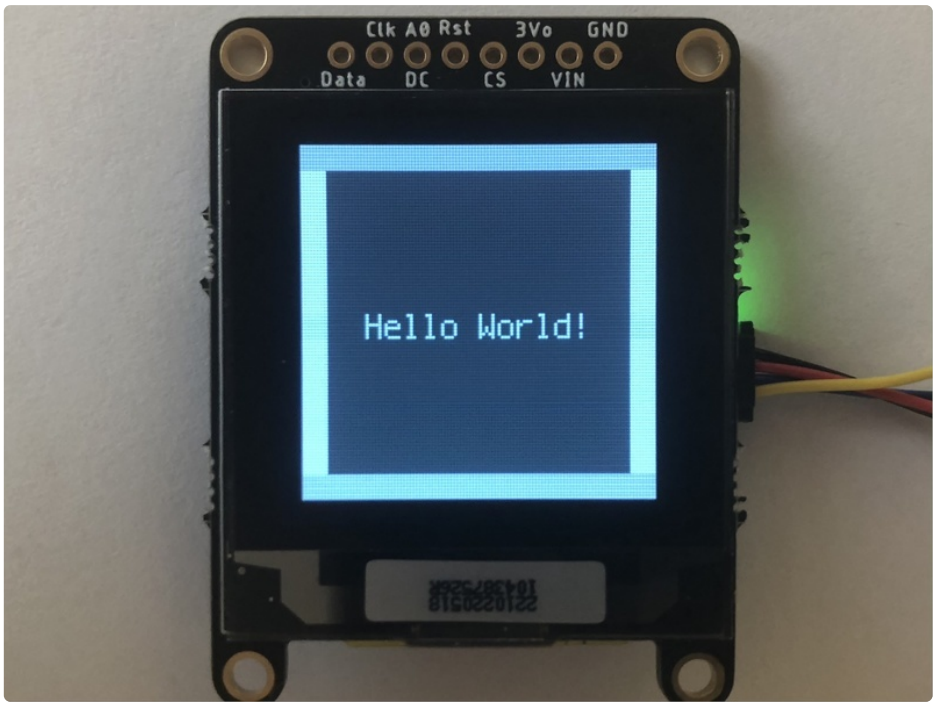
We create the label first so that we can get the width of the bounding box and multiply it by the `FONTSCALE`. This gives us the actual with of the text.

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 white text.

```
# 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

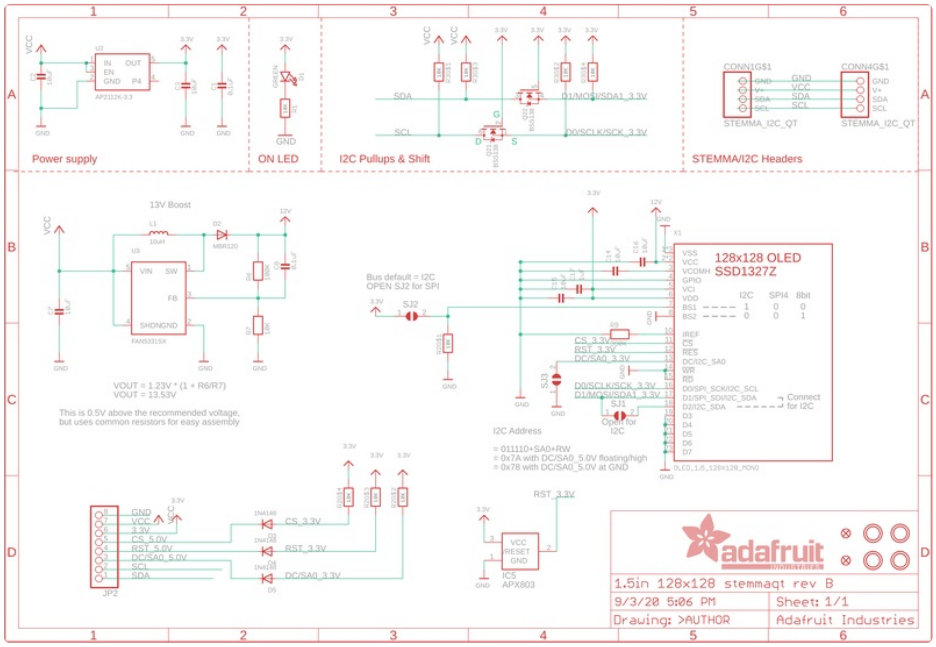
Be sure to check out this excellent [guide to CircuitPython Display Support Using displayio (https://adafru.it/EGh)](https://adafru.it/EGh)

Downloads

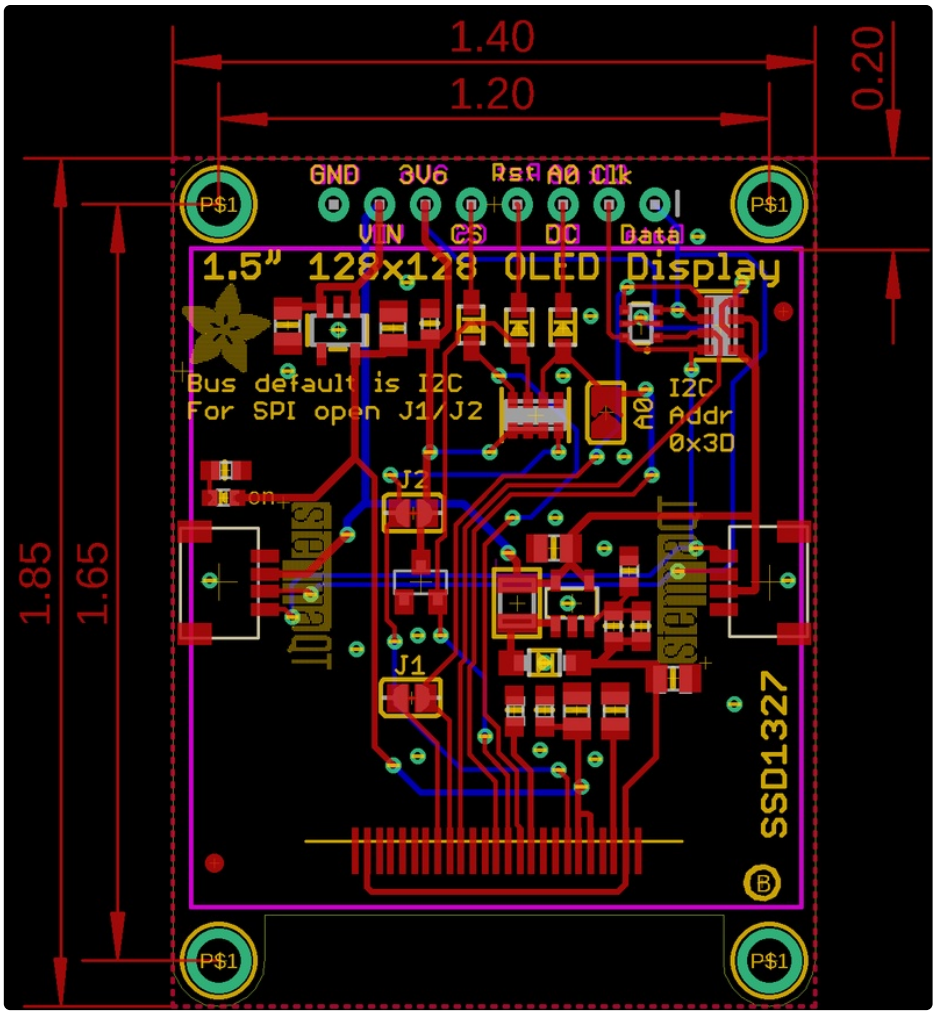
Files

- [Fritzing object in Adafruit Fritzing Library](https://adafru.it/OBn) (https://adafru.it/OBn)
- [PCB Files on GitHub](https://adafru.it/OBo) (https://adafru.it/OBo)
- [3D Models on GitHub](https://adafru.it/RcV) (https://adafru.it/RcV)

Schematic



Fab Print


1.40
1.20
0.20
1.85
1.65
GND
3V6
Rst
A0
Clk
VIN
CS
DC
Data
P$1
1.5" 128x128 OLED Display
Bus default is I2C
For SPI open J1/J2
I2C
Addr
0x3D
A0
J2
J1
SSD1327