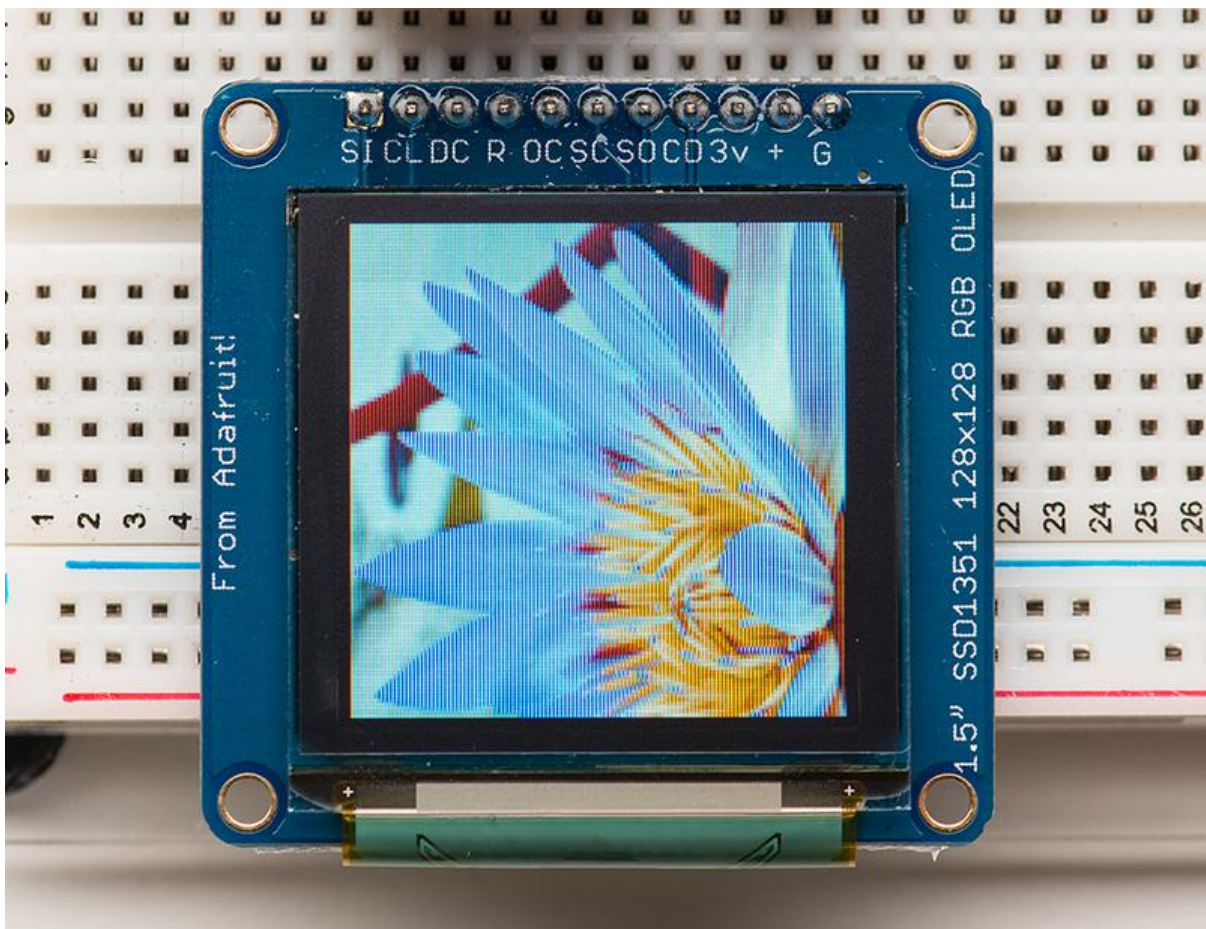




Adafruit 1.27" and 1.5" Color OLED Breakout Board

Created by Bill Earl



<https://learn.adafruit.com/adafruit-1-5-color-oled-breakout-board>

Last updated on 2023-07-05 02:07:13 PM EDT

Table of Contents

Overview	5
<ul style="list-style-type: none">• Board Technical Details	
Pinouts	7
<ul style="list-style-type: none">• EYESPI• Breakout Pins	
EYESPI	8
<ul style="list-style-type: none">• The EYESPI Connector and Cables• Wiring Your EYESPI Display• EYESPI Pins	
Plugging in an EYESPI Cable	12
Assembly	14
<ul style="list-style-type: none">• Prepare the header strip• Position the display• And Solder!• Remove the protective film	
Wiring and Graphics Test	17
<ul style="list-style-type: none">• Hint:• Installing the Arduino software	
Drawing Bitmaps	21
<ul style="list-style-type: none">• Wiring for the Bitmap Example• Hint:• Bitmap Example Sketch• Insert the card• Copy the bitmap file• Load the bitmap example sketch	
CircuitPython Displayio Quickstart	27
<ul style="list-style-type: none">• Preparing the Breakout• Required CircuitPython Libraries• Code Example Additional Libraries• CircuitPython Code Example• Where to go from here	
Python Wiring and Setup	33
<ul style="list-style-type: none">• Wiring• ILI9341 and HX-8357-based Displays• ST7789 and ST7735-based Displays• SSD1351-based Displays• SSD1331-based Display• Setup• Python Installation of RGB Display Library• DejaVu TTF Font• Pillow Library	

Python Usage

41

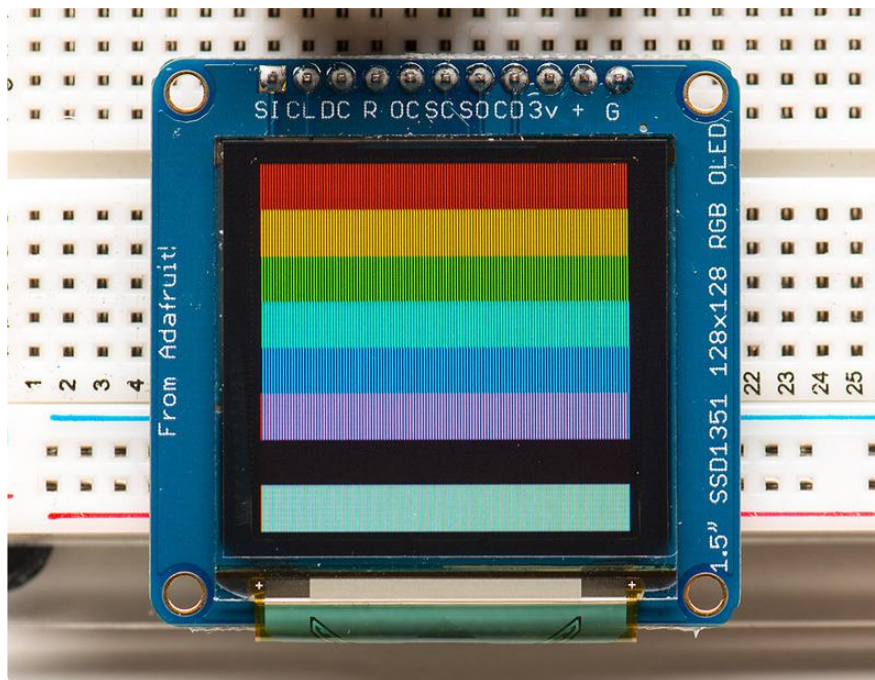
-
- [Turning on the Backlight](#)
 - [Displaying an Image](#)
 - [Drawing Shapes and Text](#)
 - [Displaying System Information](#)

Downloads and Links

54

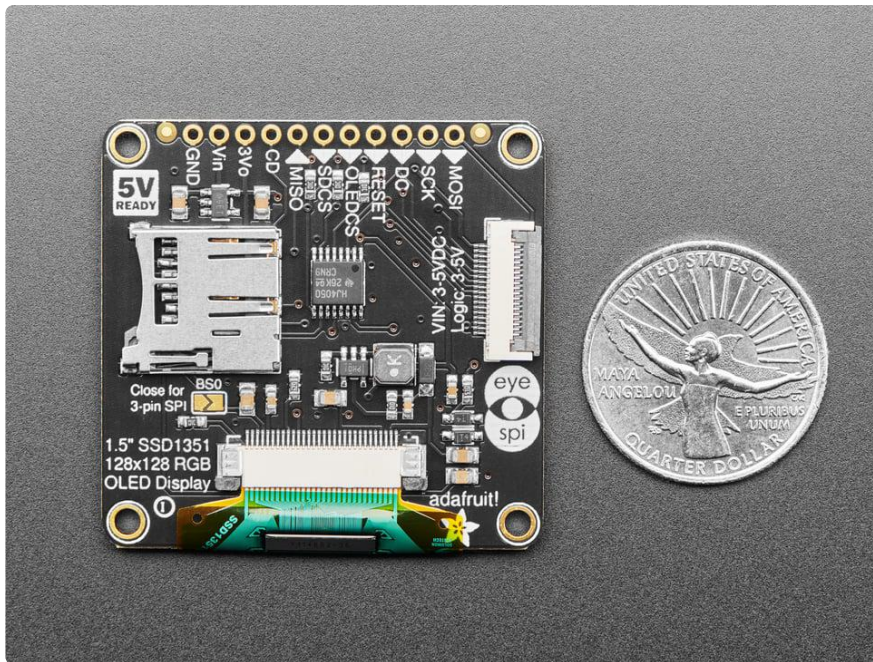
-
- [Data Sheets:](#)
 - [Schematic](#)

Overview

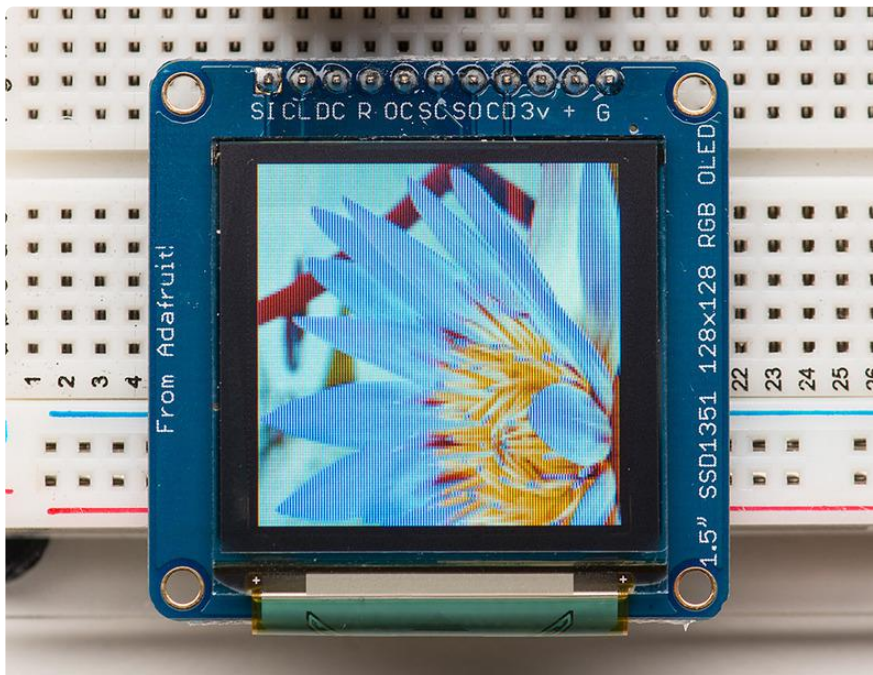


We love our black and white monochrome displays but we also like to dabble with some color now and then. Our big 1.5" color OLED displays are perfect when you need a small display with vivid, high-contrast 16-bit color. The visible portion of the OLED measures 1.5" diagonal and contains 128x128 RGB pixels, each one made of red, green and blue OLEDs. Each pixel can be set with 16-bits of resolution for a large range of colors. Because the display uses OLEDs, there is no backlight, and the contrast is very high (black is really black). We picked this display for its excellent color, this is the nicest mini OLED we could find!

This OLED uses the SSD1351 driver chip, which manages the display. You can talk to the driver chip using 4-wire write-only SPI (clock, data, chip select, data/command and an optional reset pin). Included on the fully assembled breakout is the OLED display and a small boost converter (required for providing 12V to the OLED) and a microSD card holder. This design includes built-in logic level shifting so you can use it with 3-5VDC power and logic levels. Our example code shows how to read a bitmap from the uSD card and display it all via SPI.



The 1.5" version of the display comes with an EYESPI connector! This 18-pin 0.5mm pitch FPC connector has a flip-top connector for using a flex cable to hook up your display. It enables you to avoid soldering and get your display up off of the breadboard! Consider it a sort of "STEMMA QT for displays" - a way to quickly connect and extend display wiring that uses a lot of SPI pins. It also allows for communicating with displays over longer distances. The [EYESPI flex cables](#) () are available in multiple lengths to suit any project. This is especially useful for projects where you want your display mounted separate from your microcontroller.

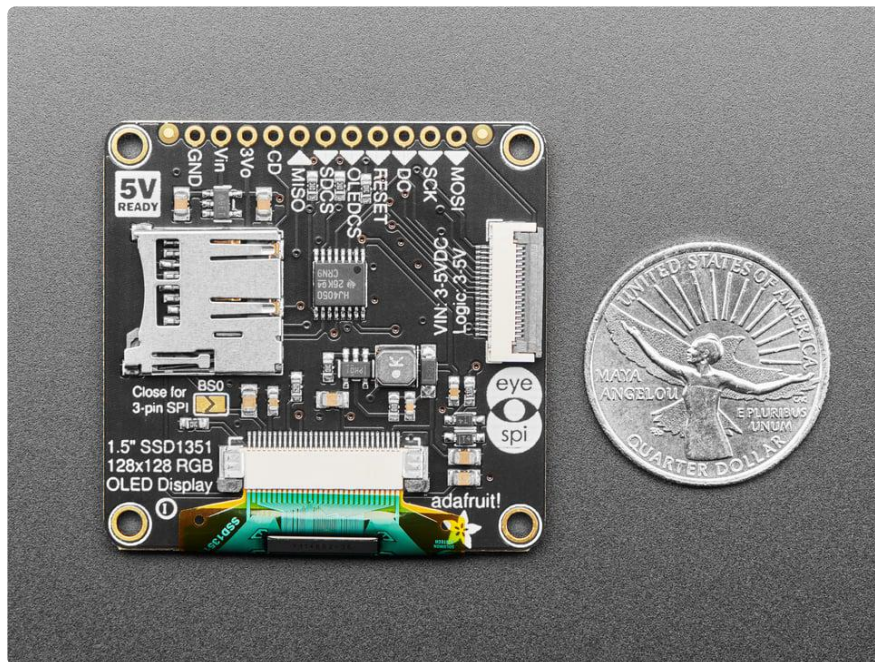


Board Technical Details

- 1.5" diagonal OLED, 16-bit color
- SPI interface
- 3.3-5V logic and power
- Micro-SD card holder
- Dimensions: 43.17mm / 1.7" x 42mm / 1.65" x 5.42mm / 0.2"

Please note: all OLEDs have a “half life” — their brightness naturally diminishes over time, albeit over many thousands of hours. This makes them a poor choice for always-on 24/7/365 use. Best to turn off the display when inactive, or consider using a color LCD for continuously running projects.

Pinouts



EYESPI

These displays now come with an EYESPI connector, which is an 18pin 0.5mm pitch connector that allows you to use a flex cable to connect your display to your microcontroller. For more details, visit the [EYESPI page \(\)](#).

Breakout Pins

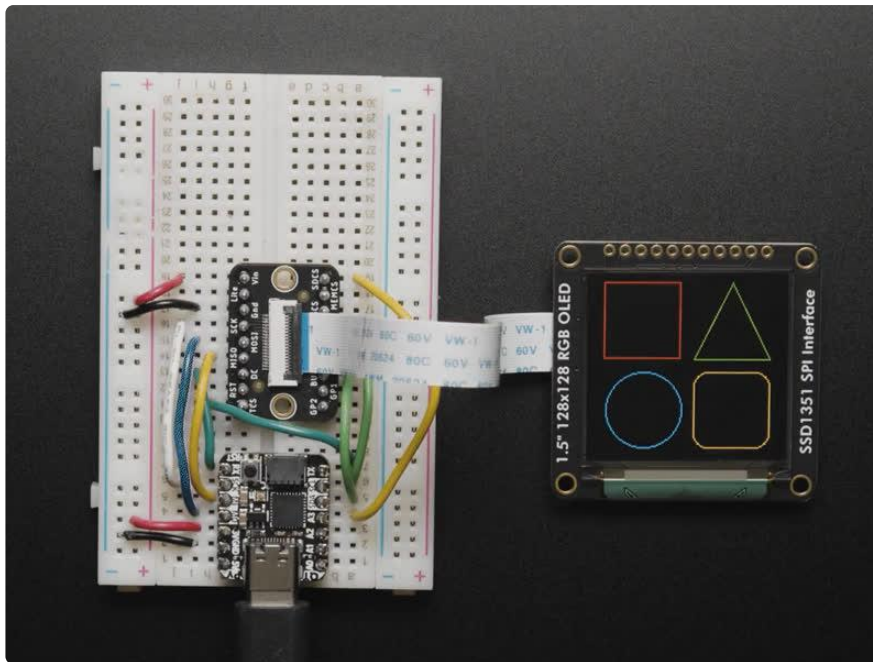
This color display uses SPI to receive image data. That means you need at least 4 pins - clock, data in, oled cs and d/c. If you'd like to have SD card usage too, add

another 2 pins - data out and card cs. However, there's a couple other pins you may want to use, lets go thru them all!

- Lite - this is the PWM input for the backlight control. Connect to 3-5VDC to turn on the backlight. Connect to ground to turn it off. Or, you can PWM at any frequency.
- MISO - this is the SPI Microcontroller In Serial Out pin, its used for the SD card. It isn't used for the OLED display which is write-only
- SCLK - this is the SPI clock input pin
- MOSI - this is the SPI Microcontroller Out Serial In pin, it is used to send data from the microcontroller to the SD card and/or OLED
- OLEDCS - this is the OLED SPI chip select pin
- SDCS - this is the SD card chip select, used if you want to read from the SD card.
- DC - this is the OLED SPI data or command selector pin
- RESET - this is the OLED reset pin. Connect to ground to reset the TFT! Its best to have this pin controlled by the library so the display is reset cleanly, but you can also connect it to the Arduino Reset pin, which works for most cases.
- CD - this is the SD card detect pin.
- 3Vo - this is the 3V output from the onboard voltage regulator
- Vin - this is the power pin, connect to 3-5VDC - it has reverse polarity protection but try to wire it right!
- GND - this is the power and signal ground pin

EYESPI

Though this page uses the 1.5" OLED display to demonstrate, the concepts and instructions apply to the 1.27" OLED display.

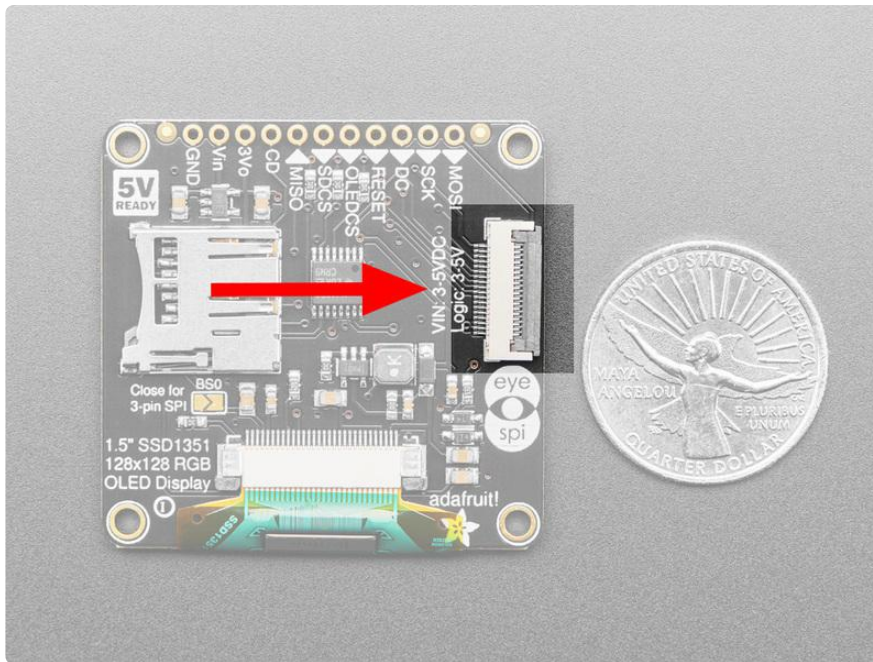


This display now comes with an EYESPI connector. This connector allows you to connect your display without soldering. There are [EYESPI cables](#) () available in multiple lengths, which means you can find one to fit any project. This is especially useful if your project requires the display to be freestanding, and not tied directly into a breadboard. Inspired by the popularity of STEMMA QT, it provides plug-n-play for displays!

The EYESPI Connector and Cables

The EYESPI connector is an 18 pin 0.5mm pitch FPC connector with a flip-top tab for locking in the associated flex cable. It is designed to allow you to connect a display, without needing to solder headers or wires to the display.

The EYESPI connector location on this display is indicated below.

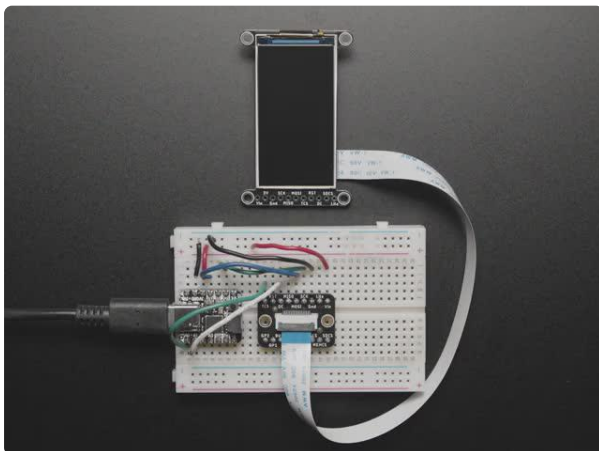


The EYESPI cables are 18 pin 0.5mm pitch flex cables. They are ~9.6mm wide, and designed to fit perfectly into the EYESPI connector. Adafruit currently offers EYESPI cables in three different lengths: [50mm](#) (), [100mm](#) (), and [200mm](#) ().

The EYESPI connector is designed to work with 18-pin 0.5mm pitch flex cables. Other flex cables, such as Raspberry Pi camera flex cables, will not work!

Wiring Your EYESPI Display

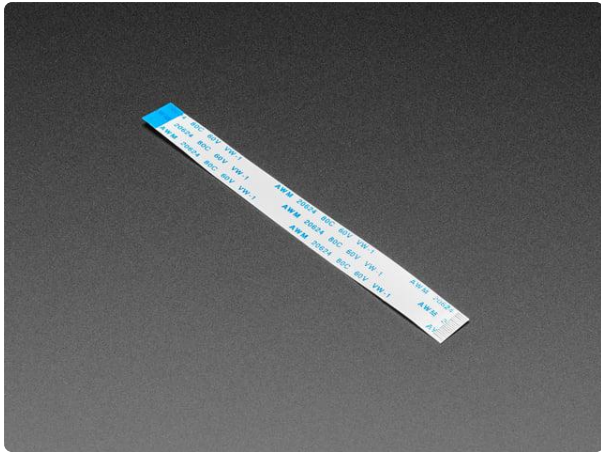
Wiring your EYESPI display to a microcontroller via the EYESPI connector requires the EYESPI breakout board and an EYESPI cable.



[Adafruit EYESPI Breakout Board - 18 Pin FPC Connector](#)

Our most recent display breakouts have come with a new feature: an 18-pin "EYESPI" standard FPC...

<https://www.adafruit.com/product/5613>



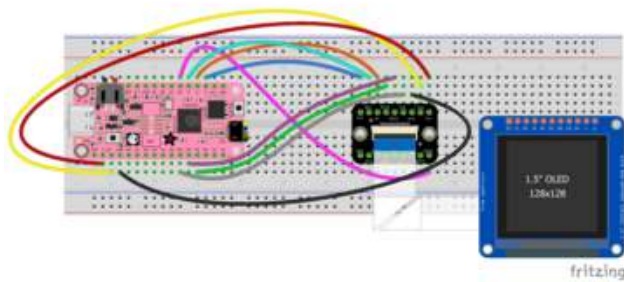
EYESPI Cable - 18 Pin 100mm long Flex PCB (FPC) A-B type

Connect this to that when a 18-pin FPC connector is needed. This 25 cm long cable is made of a flexible PCB. It's A-B style which means that pin one on one side will match...

<https://www.adafruit.com/product/5239>

The following example shows how to connect the 1.5" OLED to a Feather RP2040 using the EYESPI breakout board.

Connect the following Feather pins to the associated EYESPI breakout pins:



- breakout Vin to Feather 3.3V (red wire)
- breakout Lite to Feather 3.3V (yellow wire)
- breakout Gnd to Feather GND (black wire)
- breakout SCK to Feather SCK (grey wire)
- breakout MISO to Feather MI (green wire)
- breakout MOSI to Feather MO (purple wire)
- breakout TCS to Feather D5 (blue wire)
- breakout DC to Feather D6 (orange wire)
- breakout RST to Feather D9 (cyan wire)
- breakout SDCS to Feather D10 (pink wire)

Finally, connect your display EYESPI connector to the breakout EYESPI connector using an EYESPI cable. For details on using the EYESPI connector properly, visit [Plugging in an EYESPI Cable \(\)](#).

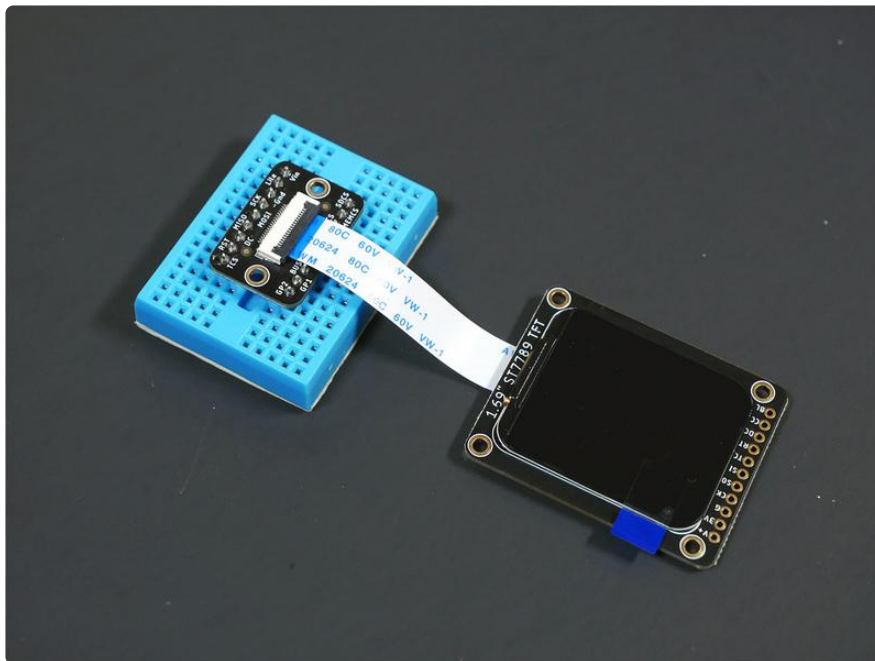
EYESPI Pins

Though there are 18 pins available on the EYESPI connector, many displays do not use all available pins. This display requires the following pins:

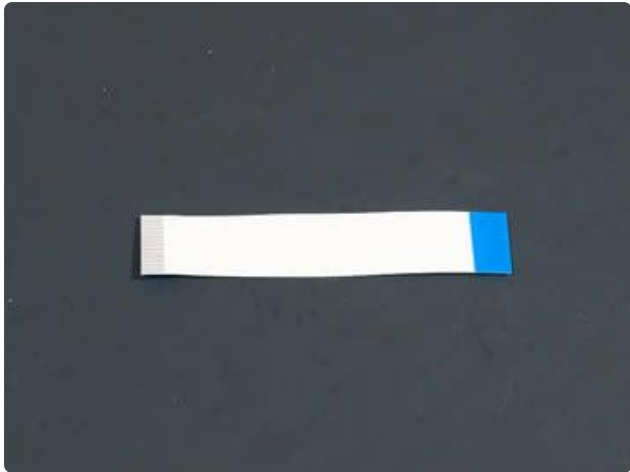
- Vin - This is the power pin. To power the board (and thus your display), connect to the same power as the logic level of your microcontroller, e.g. for a 3V micro like a Feather, use 3V, and for a 5V micro like an Arduino, use 5V.
- Lite - This is the PWM input for the backlight control. It is by default pulled high (backlight on), however, you can PWM at any frequency or pull down to turn the backlight off.

- Gnd - This is common ground for power and logic.
- MISO - This is the SPI MISO (Microcontroller In / Serial Out) pin. It's used for the SD card. It isn't used for the display because it's write-only. It is 3.3V logic out (but can be read by 5V logic).
- MOSI - This is the SPI MOSI (Microcontroller Out / Serial In) pin. It is used to send data from the microcontroller to the SD card and/or display.
- SCK - This is the SPI clock input pin.
- TCS - This is the TFT SPI chip select pin.
- RST - This is the display reset pin. Connecting to ground resets the display! It's best to have this pin controlled by the library so the display is reset cleanly, but you can also connect it to the microcontroller's Reset pin, which works for most cases. Often, there is an automatic-reset chip on the display which will reset it on power-up, making this connection unnecessary in that case.
- DC - This is the display SPI data/command selector pin.
- SDCS - This is the SD card chip select pin. This pin is required for communicating with the SD card holder onboard the connected display.

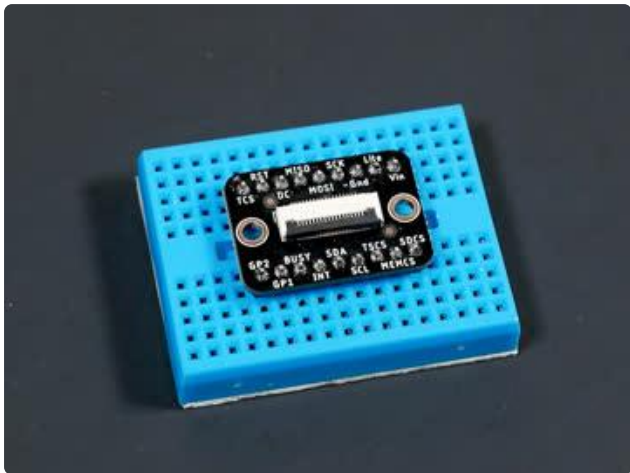
Plugging in an EYESPI Cable



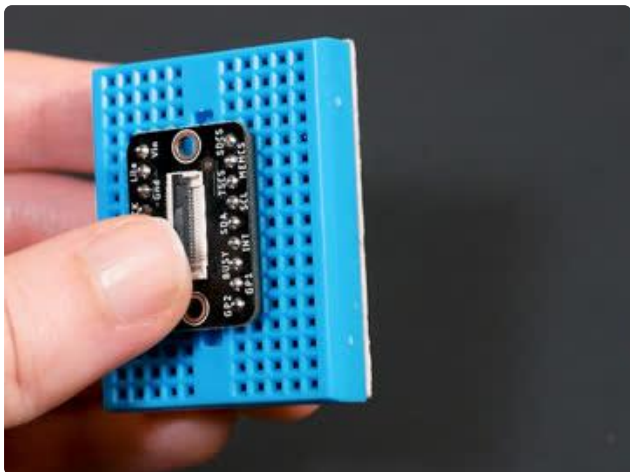
You can connect an EYESPI compatible display to the EYESPI breakout board using an EYESPI cable. An EYESPI cable is an 18 pin flexible PCB (FPC). The FPC can only be connected properly in one orientation, so be sure to follow the steps below to ensure that your display and breakout are plugged in properly.



Each EYESPI cable has blue stripes on either end. On the other side of the cable, underneath the blue stripe, are the connector pins that make contact with the FPC connector pins on the display or breakout.



To begin inserting an EYESPI cable to an FPC connector, gently lift the FPC connector black latch up.



Then, insert the EYESPI cable into the open FPC connector by sliding the cable into the connector. You want to see the blue stripe facing up towards you. This inserts the cable pins into the FPC connector.

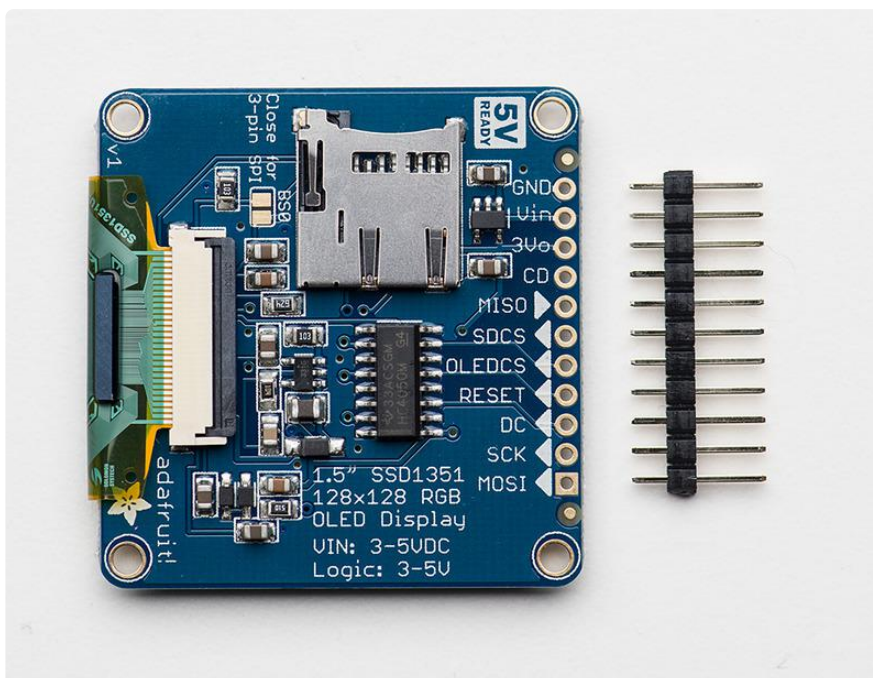


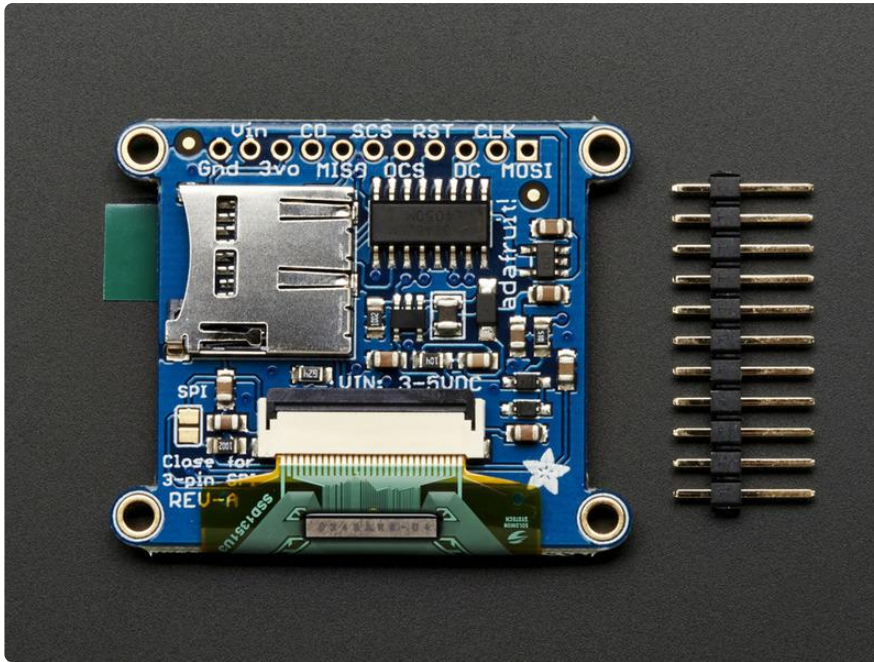
To secure the cable, lower the FPC connector latch onto the EYESPI cable.



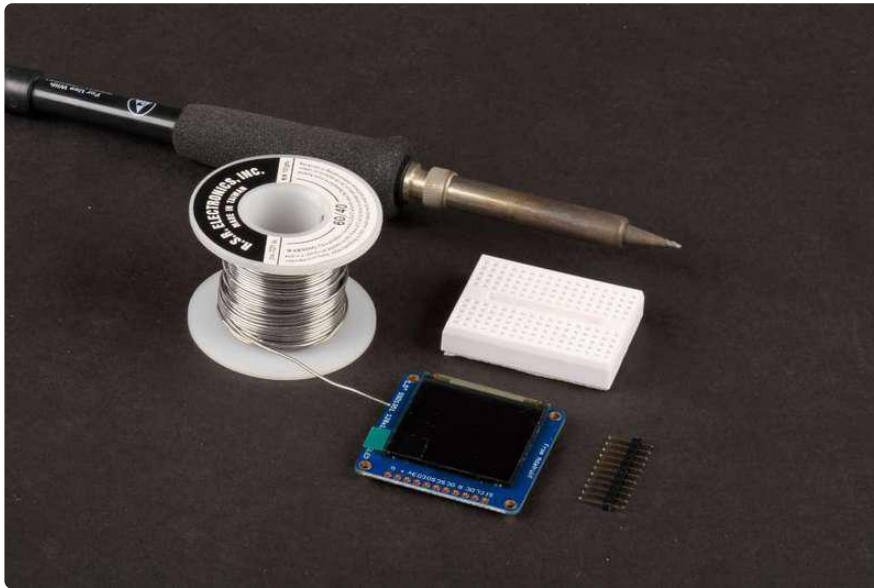
Repeat this process for the FPC connector on your display. Again, ensure that the blue stripe on either end of the cable is facing up.

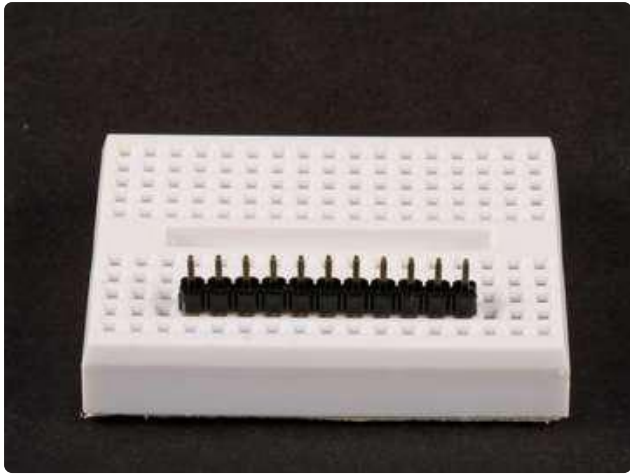
Assembly





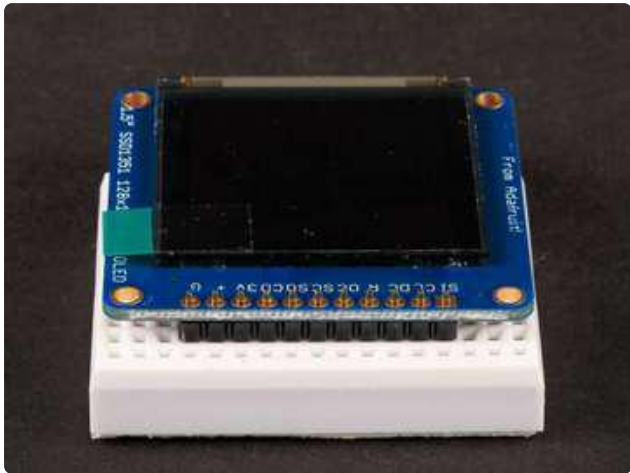
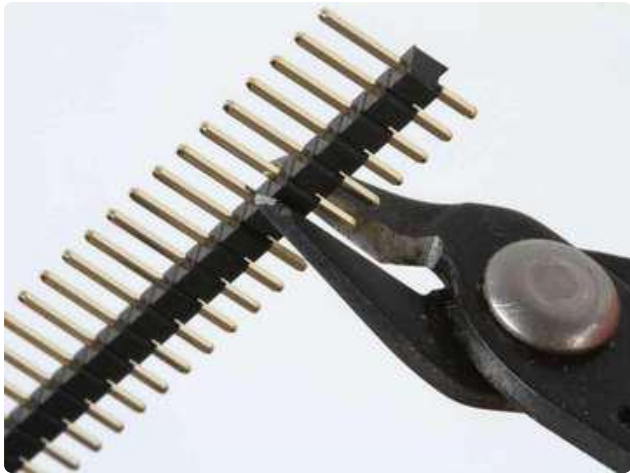
The breakout board comes fully assembled and tested. We include an optional strip of header pins to make it easier to use this display in a breadboard. The header can be installed in just a few minutes with your soldering iron:





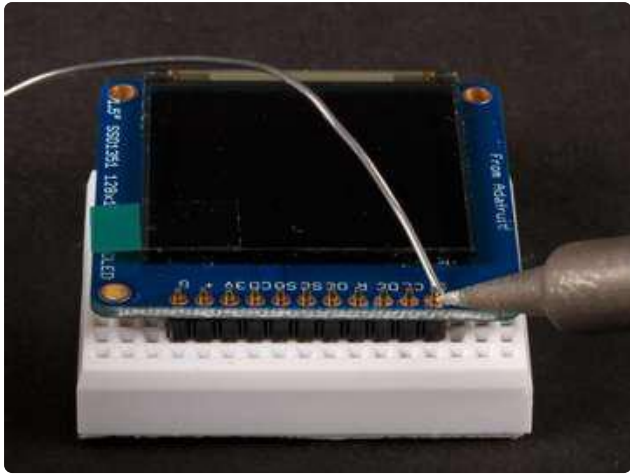
Prepare the header strip

Cut the header to size and insert (long pins down) into a breadboard to stabilize for soldering.



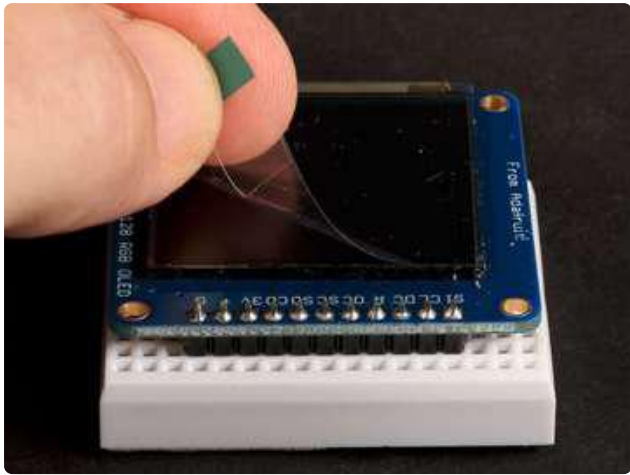
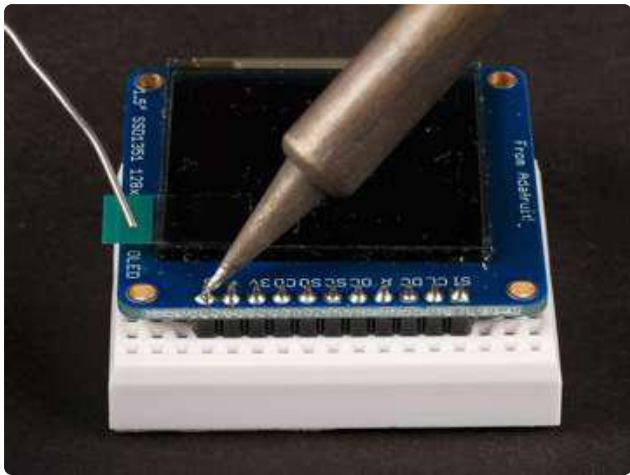
Position the display

Place the display breakout on the header so that the short pins protrude through the holes.



And Solder!

Solder all pins to assure a good electrical connection.

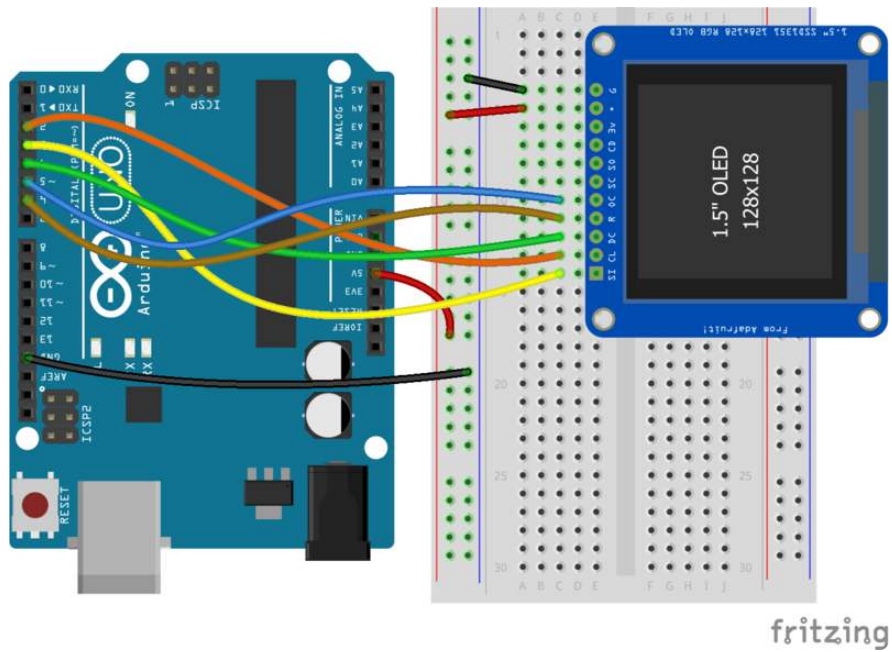


Remove the protective film

Gently pull up on the tab to remove the film.

Wiring and Graphics Test

The pinout ordering is the same for both the 1.27" and 1.5" version of the OLED!



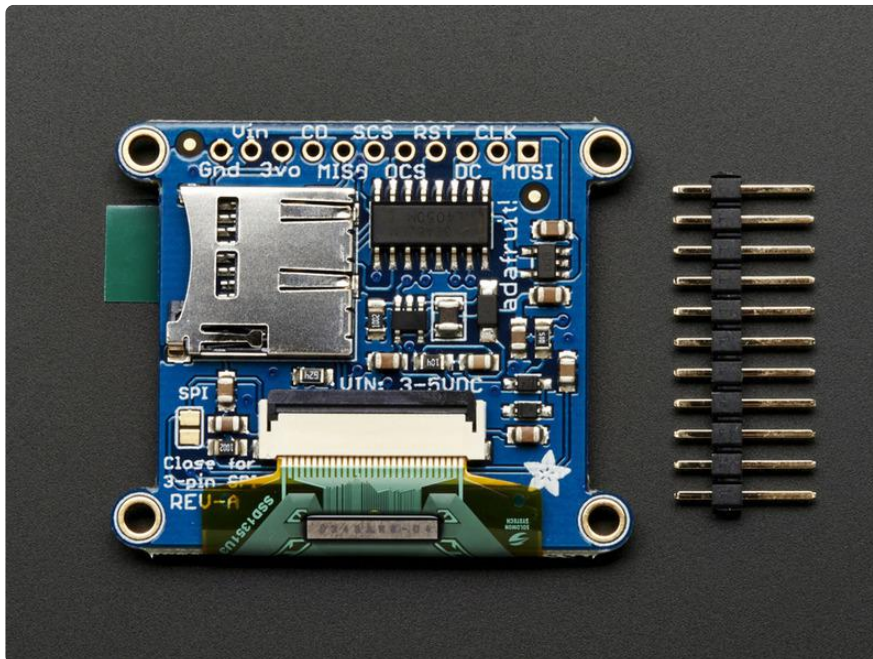
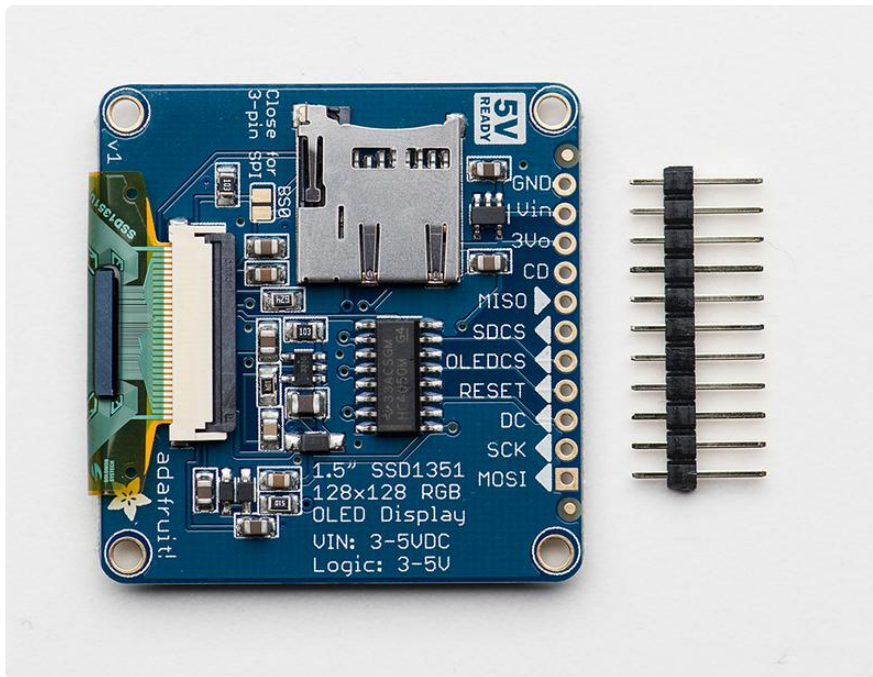
oleddemo.fzz Fritzing File

The library supports flexible wiring to minimize pin conflicts with other shields and breakouts. For the initial test, we'll use the same wiring as the "test" example from the library:

- GND -> GND (G)
- 5v -> VIN (+)
- #2 -> SCLK (CL)
- #3 -> MOSI (SI)
- #4 -> DC
- #5 -> OLEDCS (OC)
- #6 -> RST (R)

Hint:

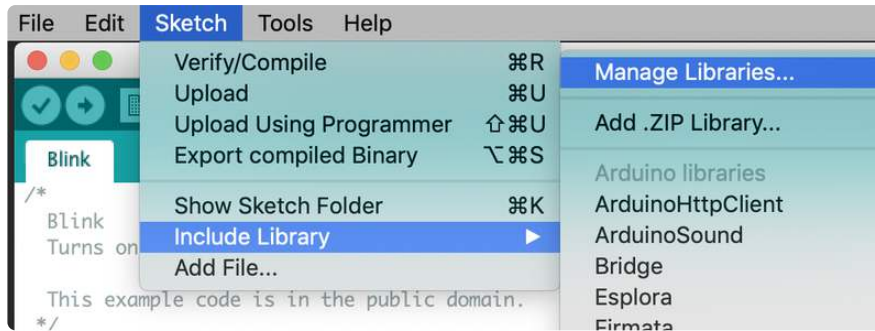
If you are confused by the abbreviations on the front of the board, the full signal names are printed on the back!



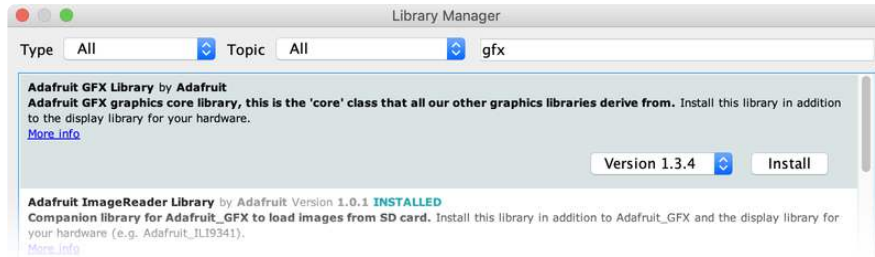
Installing the Arduino software

Now we can run the test software on the Arduino.

Three libraries need to be installed using the Arduino Library Manager...this is the preferred and modern way. From the Arduino "Sketch" menu, select "Include Library" then "Manage Libraries..."



Type “gfx” in the search field to quickly find the first library — Adafruit_GFX:



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

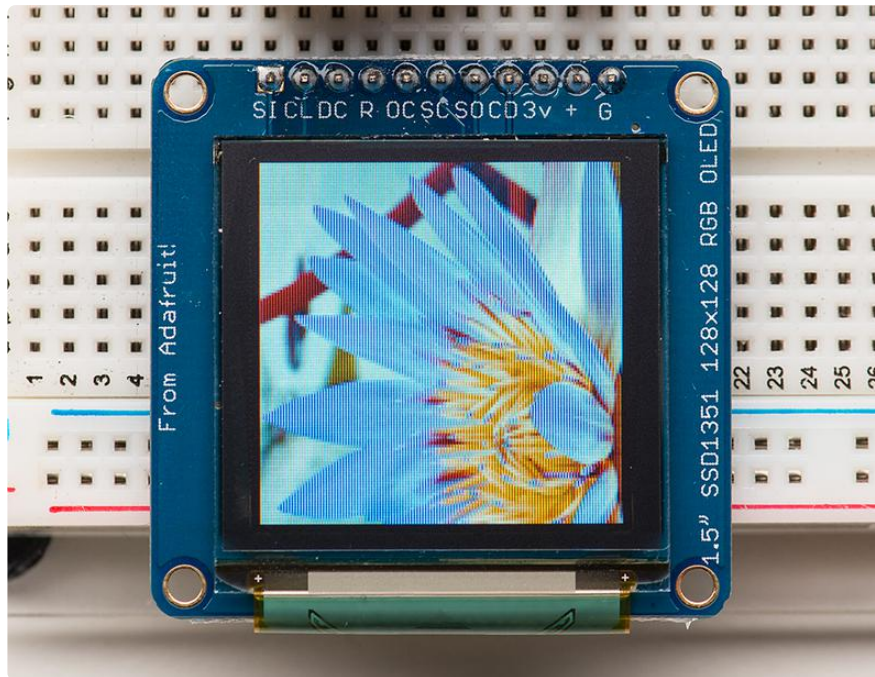
Then repeat the search and install steps, looking for the Adafruit_SSD1351 library.

After you restart, you should be able to select File→Examples→Adafruit_SSD1351→tes t - this is the example sketch that just tests the display by drawing text and shapes. Upload the sketch and you should see the following:

The test sketch demonstrates all the basic drawing functions of the Adafruit GFX Library. Read through the code to see how to draw text, circles, lines, etc.

[For a detailed tutorial on the Adafruit GFX library, including all the functions available please visit the GFX tutorial page \(\)](#)

Drawing Bitmaps

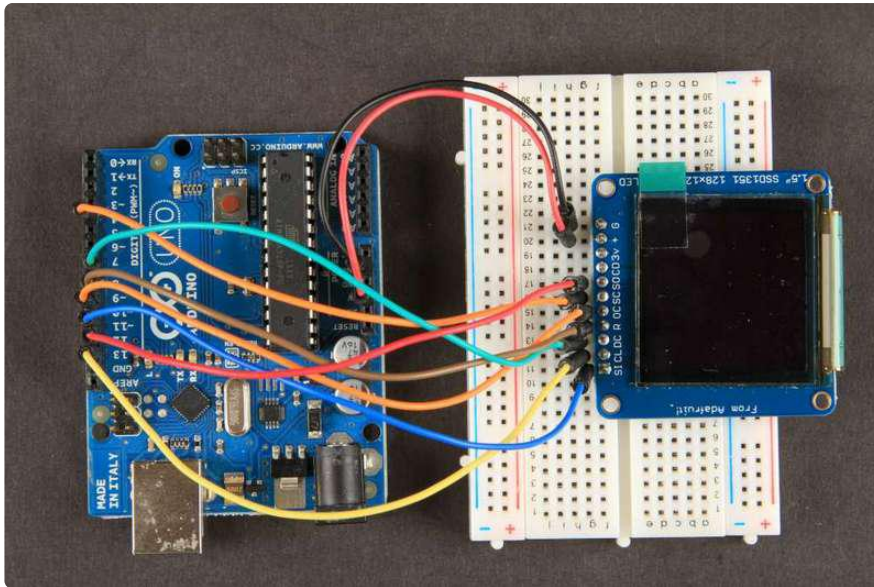


Wiring for the Bitmap Example

Drawing bitmaps from the on-board micro SD card requires a few more connections to communicate with the SD card. The library allows you to use any pins. The Arduino connections listed below match the code in the "bmp" example from the library:

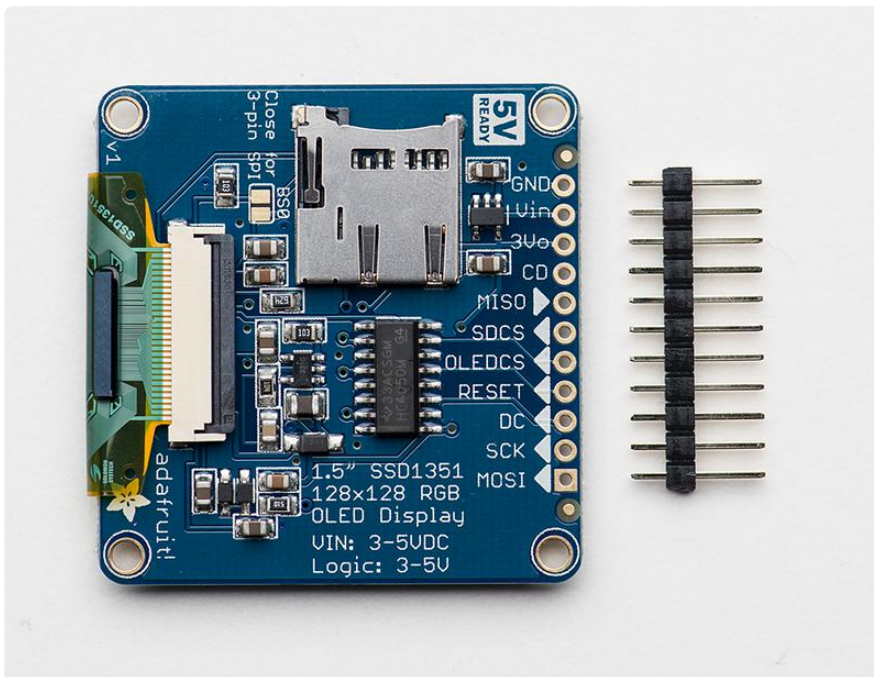
- GND -> GND (G)
- 5v -> VIN (+)
- #7 -> SDCS (SC)
- #4 -> DC
- #6 -> RST (R)
- #5 -> OLEDCS (OC)
- #11 -> MOSI (SI)
- #12 -> MISO (SO)
- #13 -> SCLK (CL)

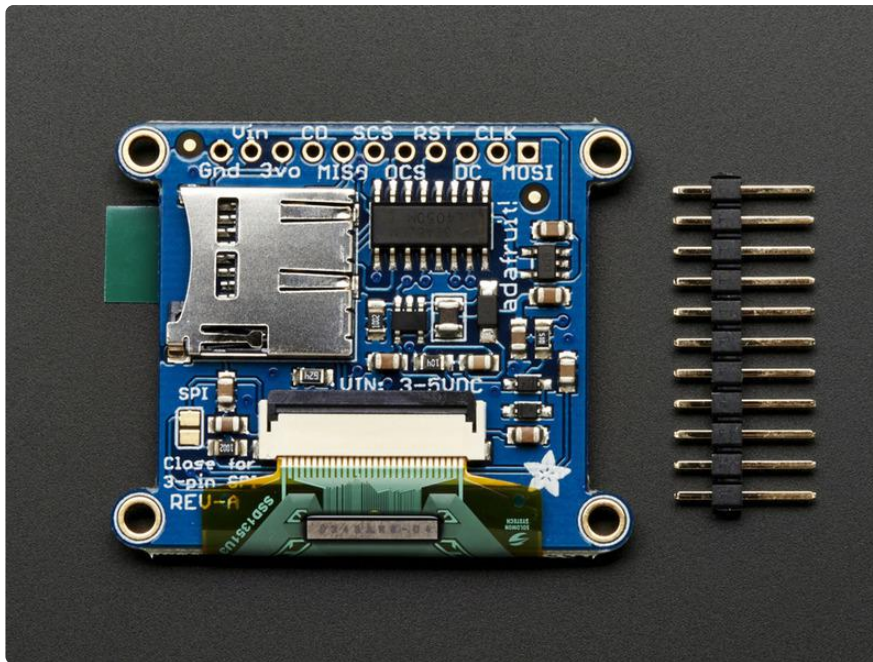
Note that the Bitmap example code uses hardware SPI wiring for maximum speed. You can still use software SPI, but make sure that the pin definitions match your wiring and that you modify the example to select the Software SPI option (#1) in the code. The SPI pins shown are for Atmega-328 processors. To use this wiring on other processors, software SPI must be used.



Hint:

If you are confused by the abbreviations on the front of the board, the full signal names are printed on the back!





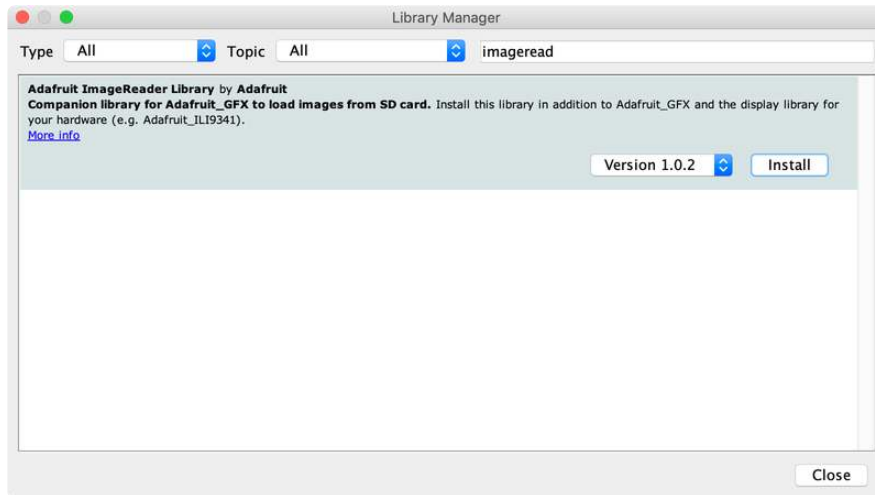
Bitmap Example Sketch

To display bitmaps from the on-board micro SD slot, you will need a [micro SD card \(http://adafruit.it/102\)](http://adafruit.it/102) formatted FAT16 or FAT32 (they almost always are by default).



There is a built in microSD card slot on the rear of the breakout and we can use that to load bitmap images!

It's really easy to draw bitmaps. We have a library for it, Adafruit_ImageReader, which can be installed through the Arduino Library Manager (Sketch→Include Library→Manage Libraries...). Enter "imageread" in the search field and the library is easy to spot:



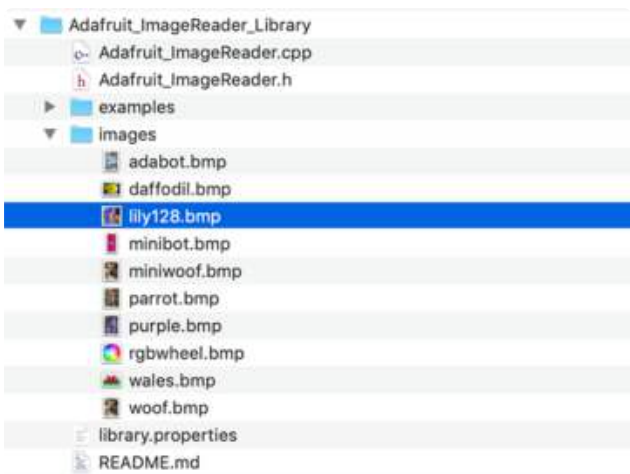
Next you can either download the image here or copy it from the images folder from inside the library files.

Download lily128.bmp



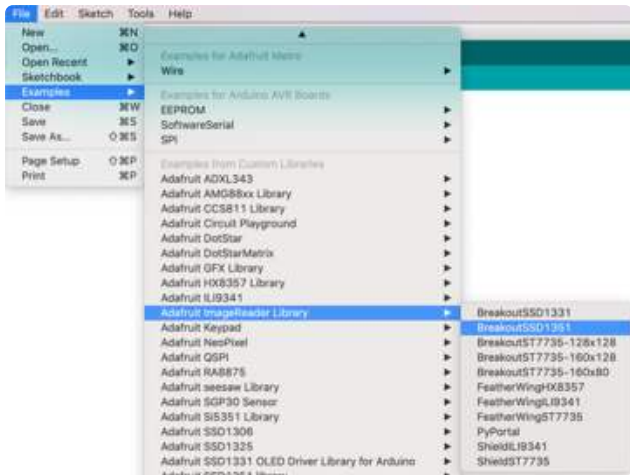
Insert the card

Insert the micro SD card into the slot on the back of the SSD1351 breakout board.



Copy the bitmap file

Copy the file "lily128.bmp" from the Adafruit_ImageReader_Library\images folder (or wherever you saved it if you downloaded the file) over to the root directory of your micro-SD card.



Load the bitmap example sketch

Select "Examples-

>Adafruit_ImageReader_Library-

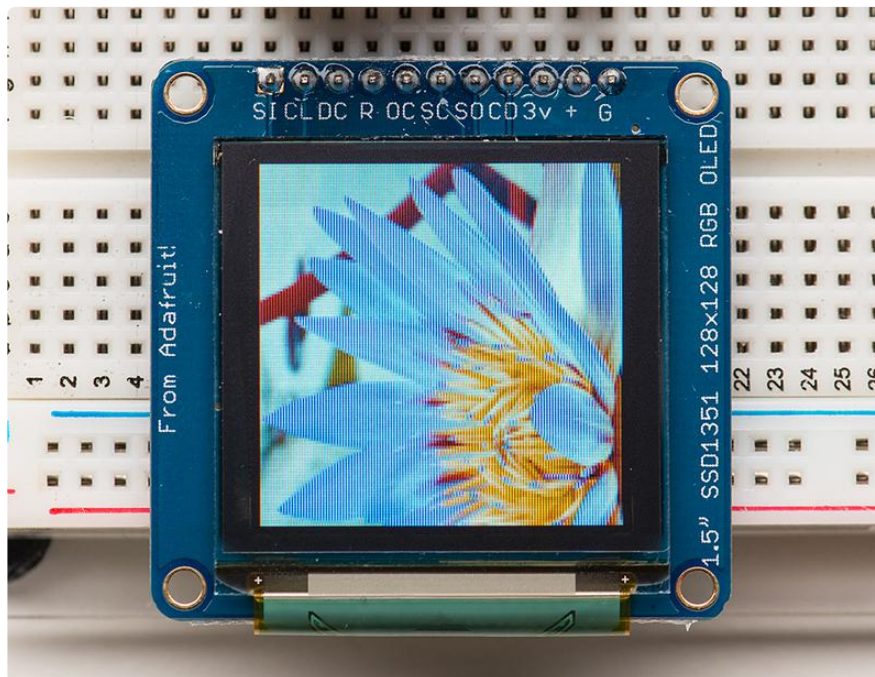
>BreakoutSSD1351" and upload it to your Arduino.

In the example, find the following section of code:

```
// Load full-screen BMP file 'rgbwheel.bmp' at position (0,0) (top left).
// Notice the 'reader' object performs this, with 'tft' as an argument.
Serial.print(F("Loading rgbwheel.bmp to screen..."));
stat = reader.drawBMP("/rgbwheel.bmp", tft, 0, 0);
reader.printStatus(stat); // How'd we do?
```

On the line with `reader.drawBMP()` change `"/rgbwheel.bmp"` to `"/lily128.bmp"`.

After that, upload it to your Arduino. When the Arduino restarts, you should see the flower as below!



To make new bitmaps, make sure they are less than 128 by 128 pixels and save them in 24-bit BMP format! They must be in 24-bit format, even if they are not 24-bit color

as that is the easiest format for the Arduino to decode. You can rotate images using the `setRotation()` procedure.

The BreakoutSSD1351 example sketch shows everything you need to work with BMP images. Here's just the vital bits broken out...

Several header files are included at the top of the sketch. All of these are required... they let us access the SD card and the display, and provide the image-reading functions:

```
#include <SPI.h>;
#include <SD.h>;
#include <Adafruit_GFX.h>;           // Core graphics library
#include <Adafruit_SSD1351.h>;      // Hardware-specific library
#include <Adafruit_ImageReader.h>;  // Image-reading functions
```

Several `#defines` relate to hardware pin numbers, all fixed values when using the shield.

Then we declare the tft screen object, and the image-reader object like so:

```
#define SD_CS    7 // SD card select pin
#define TFT_CS   5 // TFT select pin
#define TFT_DC   4 // TFT display/command pin
#define TFT_RST  6 // Or set to -1 and connect to Arduino RESET pin

Adafruit_SSD1351 tft = Adafruit_SSD1351(SCREEN_WIDTH, SCREEN_HEIGHT, &SPI,
TFT_CS, TFT_DC, TFT_RST);

Adafruit_ImageReader reader;    // Class w/image-reading functions
```

After the SD and TFT's `begin()` functions have been called (see the example sketch again, in the `setup()` function), you can then call `reader.drawBMP()` to load an image from the card to the screen:

```
ImageReturnCode stat; // Status from image-reading functions
stat = reader.drawBMP("/lily128.bmp", tft, 0, 0);
```

You can draw as many images as you want — though remember the names must be less than 8 characters long. Call like so:

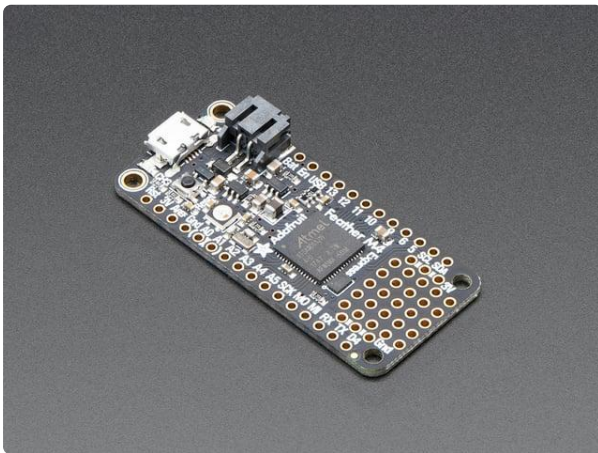
```
reader.drawBMP(filename, tft, x, y);
```

'x' and 'y' are pixel coordinates where top-left corner of the image will be placed. Images can be placed anywhere on screen...even partially off screen, the library will clip the section to load.

[Image loading is explained in greater depth in the Adafruit_GFX library guide. \(\)](#)

CircuitPython Displayio Quickstart

You will need a board capable of running CircuitPython such as the Metro M0 Express or the Metro M4 Express. You can also use boards such as the Feather M0 Express or the Feather M4 Express. We recommend either the Metro M4 or the Feather M4 Express because it's much faster and works better for driving a display. For this guide, we will be using a Feather M4 Express. The steps should be about the same for the Feather M0 Express or either of the Metros. If you haven't already, be sure to check out our [Feather M4 Express \(\)](#) guide.



[Adafruit Feather M4 Express - Featuring ATSAMD51](#)

It's what you've been waiting for, the Feather M4 Express featuring ATSAMD51. This Feather is fast like a swift, smart like an owl, strong like a ox-bird (it's half ox,... <https://www.adafruit.com/product/3857>

Preparing the Breakout

Before using the TFT Breakout, you will need to solder the headers or some wires to it. Be sure to check out the [Adafruit Guide To Excellent Soldering \(\)](#). After that the breakout should be ready to go.

Required CircuitPython Libraries

To use this display with `displayio`, there is only one required library.

`Adafruit_CircuitPython_SSD1351`

First, make sure you are running the [latest version 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 introduction guide has [a great page on how to install the library bundle \(\)](#) for both express and non-express boards.

Remember for non-express boards, you'll need to manually install the necessary libraries from the bundle:

- `adafruit_ssd1351`

Before continuing make sure your board's lib folder or root filesystem has the `adafruit_ssd1351` file copied over.

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 Code Example

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

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

import board
import terminalio
import displayio
from adafruit_display_text import label
from adafruit_ssd1351 import SSD1351

# Release any resources currently in use for the displays
displayio.release_displays()

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

display_bus = displayio.FourWire(
    spi, command=tft_dc, chip_select=tft_cs, reset=board.D9, baudrate=16000000
)

display = SSD1351(display_bus, width=128, height=128)

# Make the display context
```



```

splash = displayio.Group()
display.show(splash)

color_bitmap = displayio.Bitmap(128, 128, 1)
color_palette = displayio.Palette(1)
color_palette[0] = 0x00FF00 # Bright Green

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(108, 108, 1)
inner_palette = displayio.Palette(1)
inner_palette[0] = 0xAA0088 # Purple
inner_sprite = displayio.TileGrid(inner_bitmap, pixel_shader=inner_palette, x=10,
y=10)
splash.append(inner_sprite)

# Draw a label
text = "Hello World!"
text_area = label.Label(terminalio.FONT, text=text, color=0xFFFF00, x=30, y=64)
splash.append(text_area)

while True:
    pass

```

Let's take a look at the sections of code one by one. We're going to take a look at the code for the 1.5" display, but there is also an example available for the 128x96 1.27" display. The example is the same as we are covering in here but with the numbers adjusted for the different height.

We start by importing the board so that we can initialize `SPI`, `displayio`, `terminalio` for the font, a `label`, and the `adafruit_ssd1351` driver.

```

import board
import displayio
import terminalio
from adafruit_display_text import label
from adafruit_ssd1351 import SSD1351

```

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

```
displayio.release_displays()
```

Next, 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.

```

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

```

In the next line, we set the display bus to FourWire which makes use of the SPI bus. Additionally, we need to set the baudrate to 16MHz since that is the maximum speed that the SSD1351 chip will run at. Anything higher creates strange artifacts on the screen.

```
display_bus = displayio.FourWire(spi, command=tft_dc, chip_select=tft_cs,
                                reset=board.D9, baudrate=16000000)
```

Finally, we initialize the driver with a width of 128 and a height of 128. 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 = SSD1351(display_bus, width=128, height=128)
```



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.

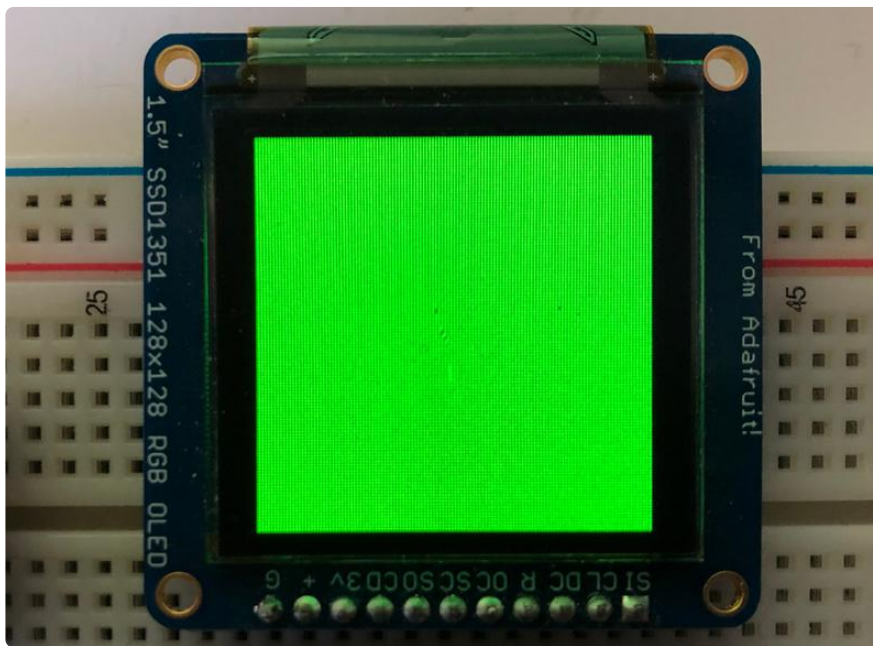
```
splash = displayio.Group(max_size=10)
display.show(splash)
```

Next we create a Bitmap which 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. The Bitmaps can currently handle up to 256 different colors. We create a Palette with one color and set that color to 0x00FF00 which happens to be green. Colors are Hexadecimal values in the format of RRGGBB. Even though the Bitmaps can only handle 256 colors at a time, you get to define what those 256 different colors are.

```
color_bitmap = displayio.Bitmap(128, 128, 1)
color_palette = displayio.Palette(1)
color_palette[0] = 0x00FF00 # Bright Green
```

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)
```

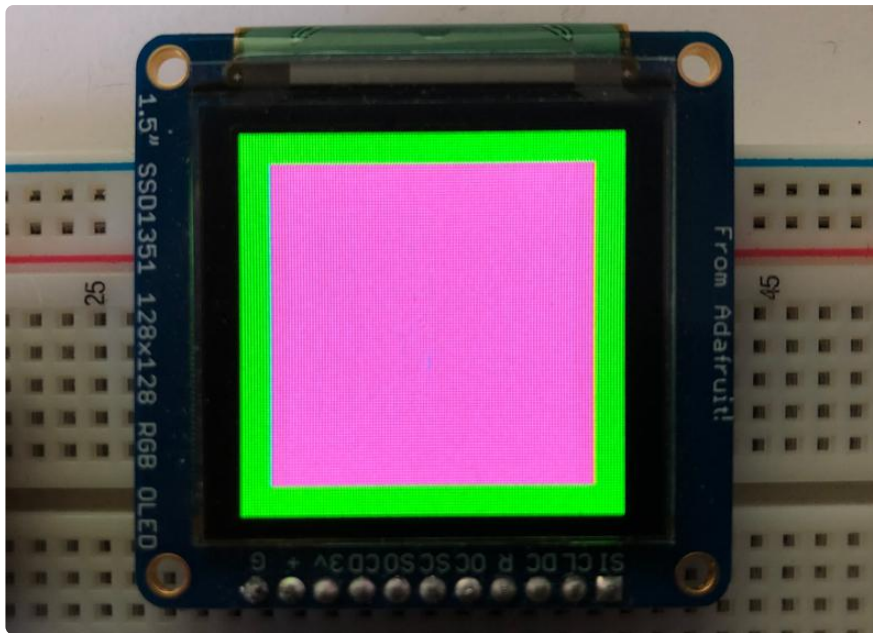


Next we will create a smaller purple square. The easiest way to do this is to create a new bitmap that is a little smaller than the full screen with a single color and place it in a specific location. In this case, we will create a bitmap that is 10 pixels smaller on each side. The screen is 128x128, so we'll want to subtract 20 from each of those numbers.

We'll also want to place it at the position `(10, 10)` so that it ends up centered.

```
inner_bitmap = displayio.Bitmap(108, 108, 1)
inner_palette = displayio.Palette(1)
inner_palette[0] = 0xAA0088 # Purple
inner_sprite = displayio.TileGrid(inner_bitmap,
                                   pixel_shader=inner_palette,
                                   x=10, y=10)
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.



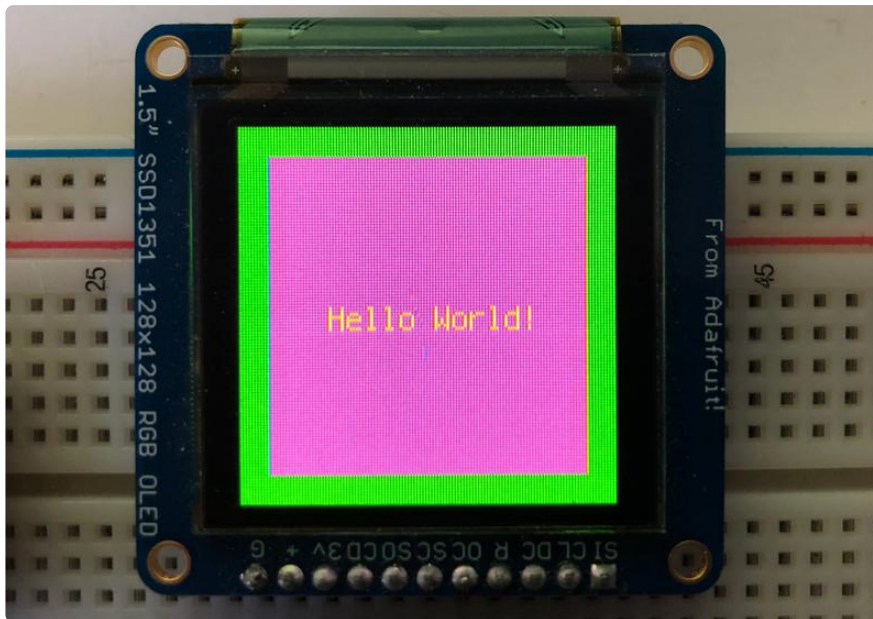
Next let's add a label that says "Hello World!" on top of that. We're going to use the built-in Terminal Font. In this example, we won't be doing any scaling because of the small resolution compared to some of the other displays, so we'll add the label directly the main group. If we were scaling, we would have used a subgroup.

Labels are centered vertically, so we'll place it at 64 for the Y coordinate, and around 30 pixels make it appear to be centered horizontally, but if you want to change the text, change this to whatever looks good to you. Let's go with some yellow text, so we'll pass it a value of `0xFFFF00`.

```
text = "Hello World!"
text_area = label.Label(terminalio.FONT, text=text, color=0xFFFF00, x=30, y=64)
splash.append(text_area)
```

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 \(\)](#)

Python Wiring and Setup

Wiring

It's easy to use display breakouts with Python and the [Adafruit CircuitPython RGB Display \(\)](#) module. This module allows you to easily write Python code to control the display.

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

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 display as shown below to your Raspberry Pi.

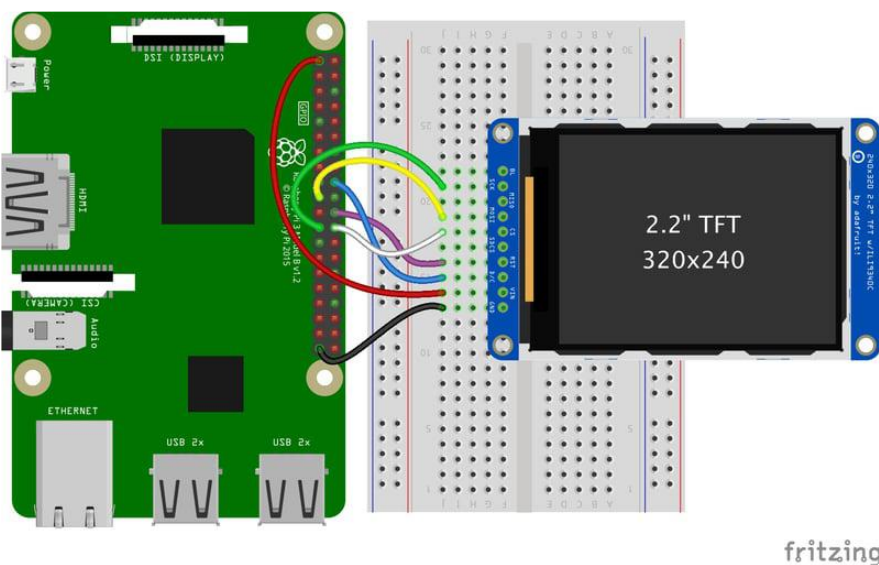
Note this is not a kernel driver that will let you have the console appear on the TFT. However, this is handy when you can't install an fbft driver, and want to use the TFT purely from 'user Python' code!

You can only use this technique with Linux/computer devices that have hardware SPI support, and not all single board computers have an SPI device so check before continuing

ILI9341 and HX-8357-based Displays

2.2" Display

- CLK connects to SPI clock. On the Raspberry Pi, that's SLCK
- MOSI connects to SPI MOSI. On the Raspberry Pi, that's also MOSI
- CS connects to our SPI Chip Select pin. We'll be using CE0
- D/C connects to our SPI Chip Select pin. We'll be using GPIO 25, but this can be changed later.
- RST connects to our Reset pin. We'll be using GPIO 24 but this can be changed later as well.
- Vin connects to the Raspberry Pi's 3V pin
- GND connects to the Raspberry Pi's ground



[Download the Fritzing Diagram](#)

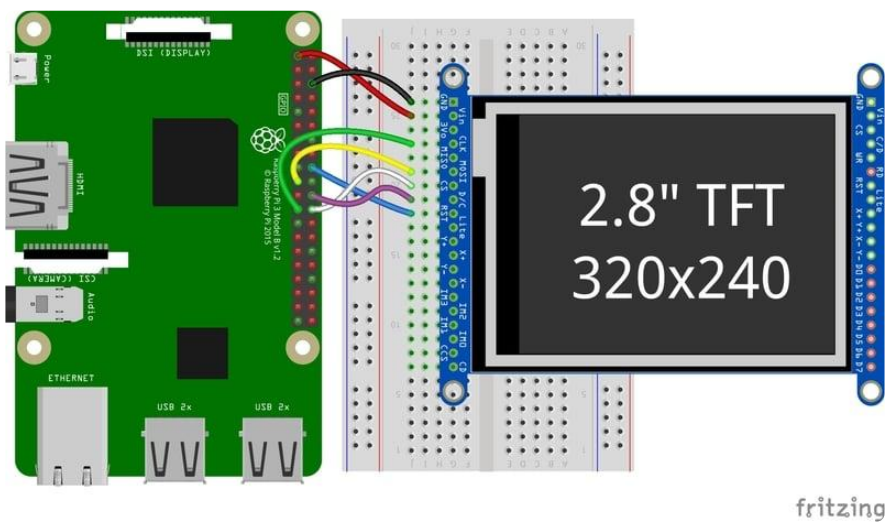
2.4", 2.8", 3.2", and 3.5" Displays

These displays are set up to use the 8-bit data lines by default. We want to use them for SPI. To do that, you'll need to either solder bridge some pads on the back or

connect the appropriate IM lines to 3.3V with jumper wires. Check the back of your display for the correct solder pads or IM lines to put it in SPI mode.

- Vin connects to the Raspberry Pi's 3V pin
- GND connects to the Raspberry Pi's ground
- CLK connects to SPI clock. On the Raspberry Pi, that's SLCK
- MOSI connects to SPI MOSI. On the Raspberry Pi, that's also MOSI
- CS connects to our SPI Chip Select pin. We'll be using CE0
- D/C connects to our SPI Chip Select pin. We'll be using GPIO 25, but this can be changed later.
- RST connects to our Reset pin. We'll be using GPIO 24 but this can be changed later as well.

These larger displays are set to use 8-bit data lines by default and may need to be modified to use SPI.



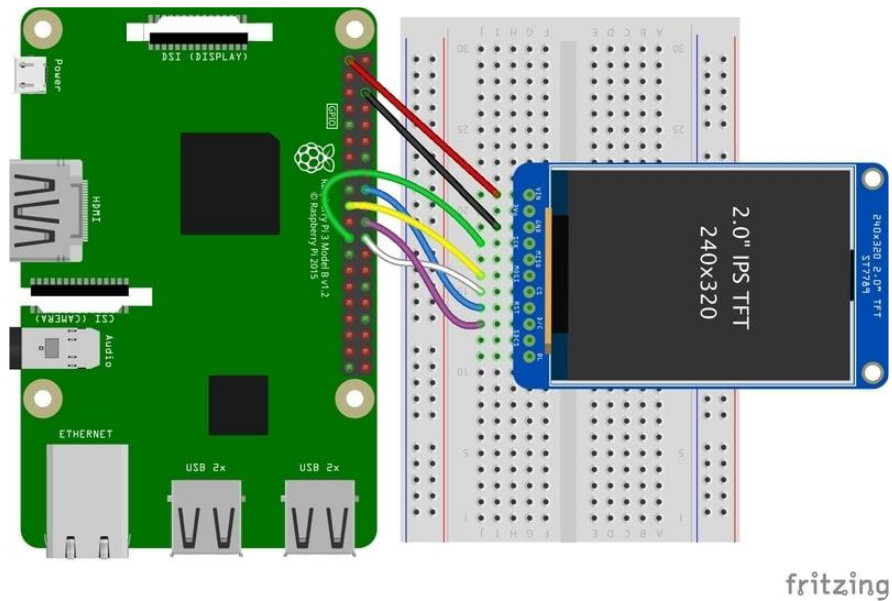
[Download the Fritzing Diagram](#)

ST7789 and ST7735-based Displays

1.3", 1.54", and 2.0" IPS TFT Display

- Vin connects to the Raspberry Pi's 3V pin
- GND connects to the Raspberry Pi's ground
- CLK connects to SPI clock. On the Raspberry Pi, that's SLCK
- MOSI connects to SPI MOSI. On the Raspberry Pi, that's also MOSI
- CS connects to our SPI Chip Select pin. We'll be using CE0
- RST connects to our Reset pin. We'll be using GPIO 24 but this can be changed later.

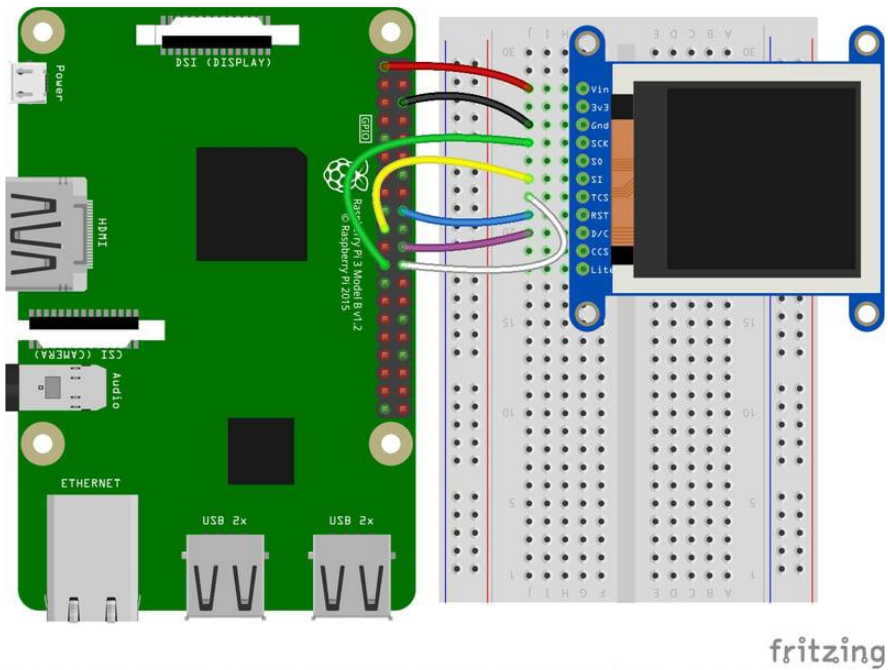
- D/C connects to our SPI Chip Select pin. We'll be using GPIO 25, but this can be changed later as well.



[Download the Fritzing Diagram](#)

0.96", 1.14", and 1.44" Displays

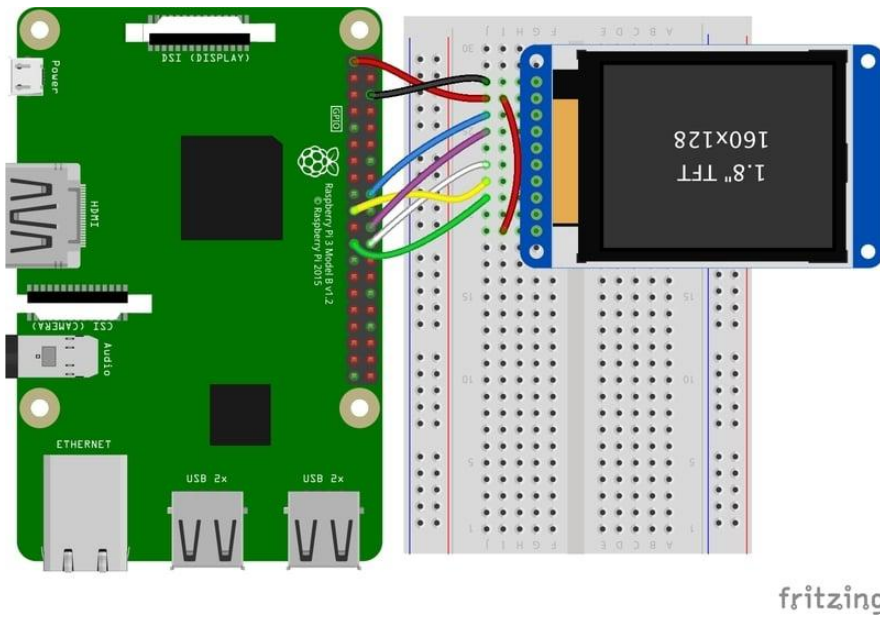
- Vin connects to the Raspberry Pi's 3V pin
- GND connects to the Raspberry Pi's ground
- CLK connects to SPI clock. On the Raspberry Pi, that's SLCK
- MOSI connects to SPI MOSI. On the Raspberry Pi, that's also MOSI
- CS connects to our SPI Chip Select pin. We'll be using CE0
- RST connects to our Reset pin. We'll be using GPIO 24 but this can be changed later.
- D/C connects to our SPI Chip Select pin. We'll be using GPIO 25, but this can be changed later as well.



[Download the Fritzing Diagram](#)

1.8" Display

- GND connects to the Raspberry Pi's ground
- Vin connects to the Raspberry Pi's 3V pin
- RST connects to our Reset pin. We'll be using GPIO 24 but this can be changed later.
- D/C connects to our SPI Chip Select pin. We'll be using GPIO 25, but this can be changed later as well.
- CS connects to our SPI Chip Select pin. We'll be using CE0
- MOSI connects to SPI MOSI. On the Raspberry Pi, that's also MOSI
- CLK connects to SPI clock. On the Raspberry Pi, that's SLCK
- LITE connects to the Raspberry Pi's 3V pin. This can be used to separately control the backlight.

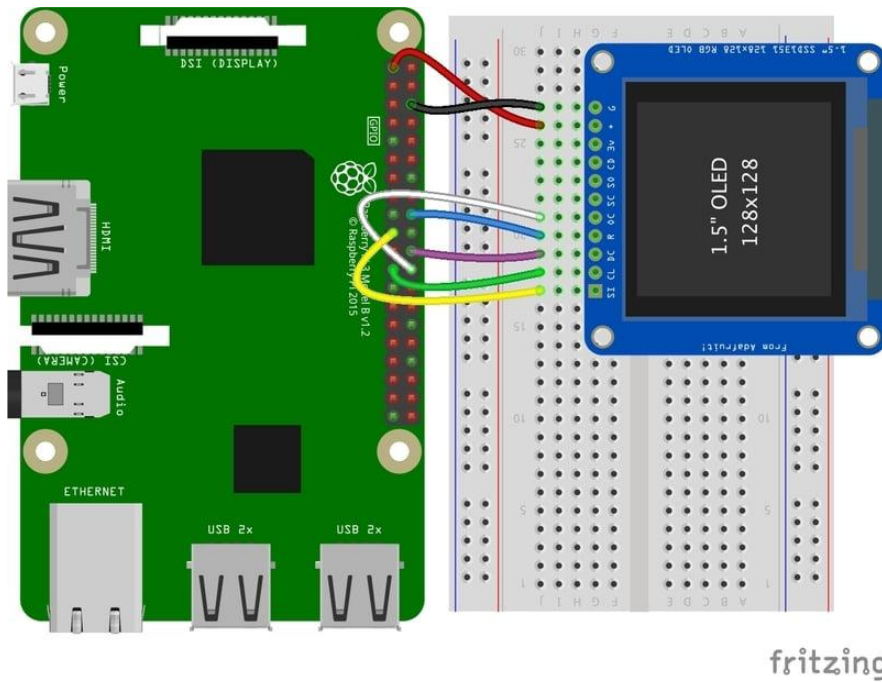


[Download the Fritzing Diagram](#)

SSD1351-based Displays

1.27" and 1.5" OLED Displays

- GND connects to the Raspberry Pi's ground
- Vin connects to the Raspberry Pi's 3V pin
- CLK connects to SPI clock. On the Raspberry Pi, that's SLCK
- MOSI connects to SPI MOSI. On the Raspberry Pi, that's also MOSI
- CS connects to our SPI Chip Select pin. We'll be using CE0
- RST connects to our Reset pin. We'll be using GPIO 24 but this can be changed later.
- D/C connects to our SPI Chip Select pin. We'll be using GPIO 25, but this can be changed later as well.

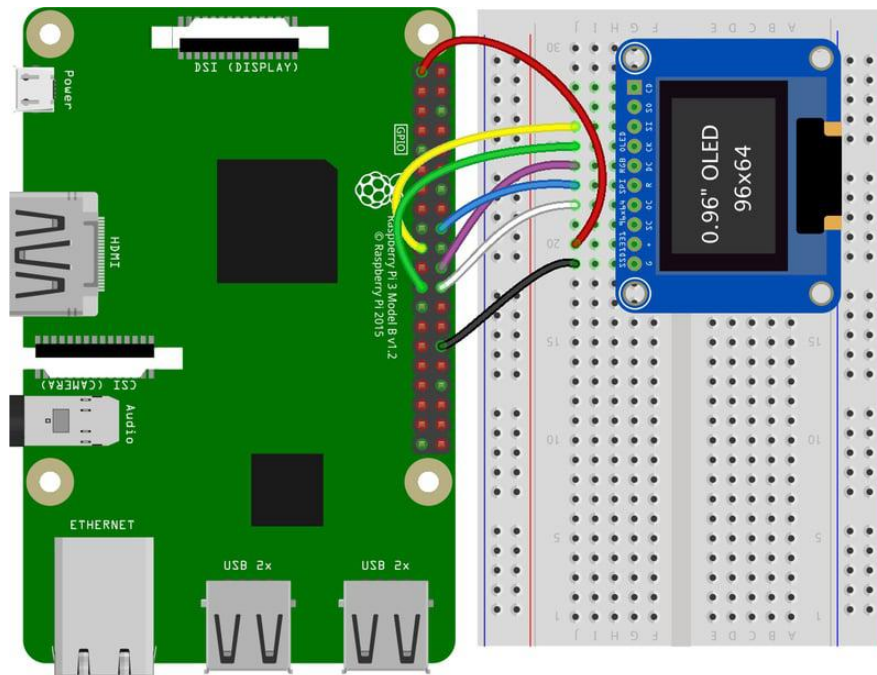


[Download the Fritzing Diagram](#)

SSD1331-based Display

0.96" OLED Display

- MOSI connects to SPI MOSI. On the Raspberry Pi, that's also MOSI
- CLK connects to SPI clock. On the Raspberry Pi, that's SLCK
- D/C connects to our SPI Chip Select pin. We'll be using GPIO 25, but this can be changed later.
- RST connects to our Reset pin. We'll be using GPIO 24 but this can be changed later as well.
- CS connects to our SPI Chip Select pin. We'll be using CE0
- Vin connects to the Raspberry Pi's 3V pin
- GND connects to the Raspberry Pi's ground



fritzing

Download the Fritzing Diagram

Setup

You'll need to install the Adafruit_Blinka library that provides the CircuitPython support in Python. This may also require enabling SPI 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 \(\)!](#)

If you have previously installed the Kernel Driver with the PiTFT Easy Setup, you will need to remove it first in order to run this example.

Python Installation of RGB Display Library

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

- `sudo pip3 install adafruit-circuitpython-rgb-display`

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`

DejaVu TTF Font

Raspberry Pi usually comes with the DejaVu font already installed, but in case it didn't, you can run the following to install it:

- `sudo apt-get install fonts-dejavu`

This package was previously calls ttf-dejavu, so if you are running an older version of Raspberry Pi OS, it may be called that.

Pillow Library

We also need PIL, the Python Imaging Library, to allow graphics and 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`

If you installed the PIL through PIP, you may need to install some additional libraries:

- `sudo apt-get install libopenjp2-7 libtiff5 libatlas-base-dev`

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

Python Usage

If you have previously installed the Kernel Driver with the PiTFT Easy Setup, you will need to remove it first in order to run this example.

Now that you have everything setup, we're going to look over three different examples. For the first, we'll take a look at automatically scaling and cropping an image and then centering it on the display.

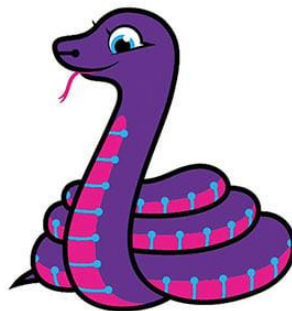
Turning on the Backlight

On some displays, the backlight is controlled by a separate pin such as the 1.3" TFT Bonnet with Joystick. On such displays, running the below code will likely result in the display remaining black. To turn on the backlight, you will need to add a small snippet of code. If your backlight pin number differs, be sure to change it in the code:

```
# Turn on the Backlight
backlight = DigitalInOut(board.D26)
backlight.switch_to_output()
backlight.value = True
```

Displaying an Image

Here's the full code to the example. We will go through it section by section to help you better understand what is going on. Let's start by downloading an image of Blinky. This image has enough border to allow resizing and cropping with a variety of display sizes and rations to still look good.



Make sure you save it as blinka.jpg and place it in the same folder as your script. Here's the code we'll be loading onto the Raspberry Pi. We'll go over the interesting parts.

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

"""
Be sure to check the learn guides for more usage information.

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)!

Author(s): Melissa LeBlanc-Williams for Adafruit Industries
```

```

"""

import digitalio
import board
from PIL import Image, ImageDraw
from adafruit_rgb_display import ili9341
from adafruit_rgb_display import st7789 # pylint: disable=unused-import
from adafruit_rgb_display import hx8357 # pylint: disable=unused-import
from adafruit_rgb_display import st7735 # pylint: disable=unused-import
from adafruit_rgb_display import ssd1351 # pylint: disable=unused-import
from adafruit_rgb_display import ssd1331 # pylint: disable=unused-import

# Configuration for CS and DC pins (these are PiTFT defaults):
cs_pin = digitalio.DigitalInOut(board.CE0)
dc_pin = digitalio.DigitalInOut(board.D25)
reset_pin = digitalio.DigitalInOut(board.D24)

# Config for display baudrate (default max is 24mhz):
BAUDRATE = 24000000

# Setup SPI bus using hardware SPI:
spi = board.SPI()

# pylint: disable=line-too-long
# Create the display:
# disp = st7789.ST7789(spi, rotation=90, # 2.0" ST7789
# disp = st7789.ST7789(spi, height=240, y_offset=80, rotation=180, # 1.3", 1.54"
ST7789
# disp = st7789.ST7789(spi, rotation=90, width=135, height=240, x_offset=53,
y_offset=40, # 1.14" ST7789
# disp = st7789.ST7789(spi, rotation=90, width=172, height=320, x_offset=34, #
1.47" ST7789
# disp = st7789.ST7789(spi, rotation=270, width=170, height=320, x_offset=35, #
1.9" ST7789
# disp = hx8357.HX8357(spi, rotation=180, # 3.5" HX8357
# disp = st7735.ST7735R(spi, rotation=90, # 1.8" ST7735R
# disp = st7735.ST7735R(spi, rotation=270, height=128, x_offset=2, y_offset=3, #
1.44" ST7735R
# disp = st7735.ST7735R(spi, rotation=90, bgr=True, width=80, # 0.96" MiniTFT
Rev A ST7735R
# disp = st7735.ST7735R(spi, rotation=90, invert=True, width=80, # 0.96" MiniTFT
Rev B ST7735R
# x_offset=26, y_offset=1,
# disp = ssd1351.SSD1351(spi, rotation=180, # 1.5" SSD1351
# disp = ssd1351.SSD1351(spi, height=96, y_offset=32, rotation=180, # 1.27" SSD1351
# disp = ssd1331.SSD1331(spi, rotation=180, # 0.96" SSD1331
disp = ili9341.ILI9341(
    spi,
    rotation=90, # 2.2", 2.4", 2.8", 3.2" ILI9341
    cs=cs_pin,
    dc=dc_pin,
    rst=reset_pin,
    baudrate=BAUDRATE,
)
# pylint: enable=line-too-long

# Create blank image for drawing.
# Make sure to create image with mode 'RGB' for full color.
if disp.rotation % 180 == 90:
    height = disp.width # we swap height/width to rotate it to landscape!
    width = disp.height
else:
    width = disp.width # we swap height/width to rotate it to landscape!
    height = disp.height
image = Image.new("RGB", (width, height))

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

```

```

# Draw a black filled box to clear the image.
draw.rectangle((0, 0, width, height), outline=0, fill=(0, 0, 0))
disp.image(image)

image = Image.open("blinka.jpg")

# Scale the image to the smaller screen dimension
image_ratio = image.width / image.height
screen_ratio = width / height
if screen_ratio < image_ratio:
    scaled_width = image.width * height // image.height
    scaled_height = height
else:
    scaled_width = width
    scaled_height = image.height * width // image.width
image = image.resize((scaled_width, scaled_height), Image.Resampling.BICUBIC)

# Crop and center the image
x = scaled_width // 2 - width // 2
y = scaled_height // 2 - height // 2
image = image.crop((x, y, x + width, y + height))

# Display image.
disp.image(image)

```

So we start with our usual imports including a couple of Pillow modules and the display drivers. That is followed by defining a few pins here. The reason we chose these is because they allow you to use the same code with the PiTFT if you chose to do so.

```

import digitalio
import board
from PIL import Image, ImageDraw
import adafruit_rgb_display.ili9341 as ili9341
import adafruit_rgb_display.st7789 as st7789
import adafruit_rgb_display.hx8357 as hx8357
import adafruit_rgb_display.st7735 as st7735
import adafruit_rgb_display.ssd1351 as ssd1351
import adafruit_rgb_display.ssd1331 as ssd1331

# Configuration for CS and DC pins
cs_pin = digitalio.DigitalInOut(board.CE0)
dc_pin = digitalio.DigitalInOut(board.D25)
reset_pin = digitalio.DigitalInOut(board.D24)

```

Next we'll set the baud rate from the default 24 MHz so that it works on a variety of displays. The exception to this is the SSD1351 driver, which will automatically limit it to 16MHz even if you pass 24MHz. We'll set up our SPI bus and then initialize the display.

We wanted to make these examples work on as many displays as possible with very few changes. The ILI9341 display is selected by default. For other displays, go ahead and comment out these lines:

```

disp = ili9341.ILI9341(
    spi,
    rotation=90, # 2.2", 2.4", 2.8", 3.2" ILI9341

```


and uncomment the line appropriate for your display and possibly the line below in the case of longer initialization sequences. The displays have a rotation property so that it can be set in just one place.

```
#disp = st7789.ST7789(spi, rotation=90, # 2.0" ST7789
#disp = st7789.ST7789(spi, height=240, y_offset=80, rotation=180, # 1.3", 1.54"
ST7789
#disp = st7789.ST7789(spi, rotation=90, width=135, height=240, x_offset=53,
y_offset=40, # 1.14" ST7789
#disp = hx8357.HX8357(spi, rotation=180, # 3.5" HX8357
#disp = st7735.ST7735R(spi, rotation=90, # 1.8" ST7735R
#disp = st7735.ST7735R(spi, rotation=270, height=128, x_offset=2, y_offset=3, #
1.44" ST7735R
#disp = st7735.ST7735R(spi, rotation=90, bgr=True, width=80, # 0.96" MiniTFT
Rev A ST7735R
#disp = st7735.ST7735R(spi, rotation=90, invert=True, width=80, # 0.96" MiniTFT
Rev B ST7735R
#x_offset=26, y_offset=1,#disp = ssd1351.SSD1351(spi,
rotation=180, # 1.5" SSD1351
#disp = ssd1351.SSD1351(spi, height=96, y_offset=32, rotation=180, # 1.27" SSD1351
#disp = ssd1331.SSD1331(spi, rotation=180, # 0.96" SSD1331
disp = ili9341.ILI9341(
    spi,
    rotation=90, # 2.2", 2.4", 2.8", 3.2" ILI9341
    cs=cs_pin,
    dc=dc_pin,
    rst=reset_pin,
    baudrate=BAUDRATE
)
```

Next we read the current rotation setting of the display and if it is 90 or 270 degrees, we need to swap the width and height for our calculations, otherwise we just grab the width and height. We will create an `image` with our dimensions and use that to create a `draw` object. The `draw` object will have all of our drawing functions.

```
# Create blank image for drawing.
# Make sure to create image with mode 'RGB' for full color.
if disp.rotation % 180 == 90:
    height = disp.width # we swap height/width to rotate it to landscape!
    width = disp.height
else:
    width = disp.width # we swap height/width to rotate it to landscape!
    height = disp.height
image = Image.new('RGB', (width, height))

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

Next we clear whatever is on the screen by drawing a black rectangle. This isn't strictly necessary since it will be overwritten by the image, but it kind of sets the stage.

```
# Draw a black filled box to clear the image.
draw.rectangle((0, 0, width, height), outline=0, fill=(0, 0, 0))
disp.image(image)
```

Next we open the Blinka image, which we've named blinka.jpg, which assumes it is in the same directory that you are running the script from. Feel free to change it if it doesn't match your configuration.

```
image = Image.open("blinka.jpg")
```

Here's where it starts to get interesting. We want to scale the image so that it matches either the width or height of the display, depending on which is smaller, so that we have some of the image to chop off when we crop it. So we start by calculating the width to height ration of both the display and the image. If the height is the closer of the dimensions, we want to match the image height to the display height and let it be a bit wider than the display. Otherwise, we want to do the opposite.

Once we've figured out how we're going to scale it, we pass in the new dimensions and using a Bicubic rescaling method, we reassign the newly rescaled image back to `image`. Pillow has quite a few different methods to choose from, but Bicubic does a great job and is reasonably fast.

```
# Scale the image to the smaller screen dimension
image_ratio = image.width / image.height
screen_ratio = width / height
if screen_ratio < image_ratio:
    scaled_width = image.width * height // image.height
    scaled_height = height
else:
    scaled_width = width
    scaled_height = image.height * width // image.width
image = image.resize((scaled_width, scaled_height), Image.BICUBIC)
```

Next we want to figure the starting x and y points of the image where we want to begin cropping it so that it ends up centered. We do that by using a standard centering function, which is basically requesting the difference of the center of the display and the center of the image. Just like with scaling, we replace the `image` variable with the newly cropped image.

```
# Crop and center the image
x = scaled_width // 2 - width // 2
y = scaled_height // 2 - height // 2
image = image.crop((x, y, x + width, y + height))
```

Finally, we take our image and display it. At this point, the image should have the exact same dimensions as the display and fill it completely.

```
disp.image(image)
```



Drawing Shapes and Text

In the next example, we'll take a look at drawing shapes and text. This is very similar to the displayio example, but it uses Pillow instead. Here's the code for that.

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

"""
This demo will draw a few rectangles onto the screen along with some text
on top of that.

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)!

Author(s): Melissa LeBlanc-Williams for Adafruit Industries
"""

import digitalio
import board
from PIL import Image, ImageDraw, ImageFont
from adafruit_rgb_display import ili9341
from adafruit_rgb_display import st7789 # pylint: disable=unused-import
from adafruit_rgb_display import hx8357 # pylint: disable=unused-import
from adafruit_rgb_display import st7735 # pylint: disable=unused-import
from adafruit_rgb_display import ssd1351 # pylint: disable=unused-import
from adafruit_rgb_display import ssd1331 # pylint: disable=unused-import

# First define some constants to allow easy resizing of shapes.
BORDER = 20
FONTSIZE = 24

# Configuration for CS and DC pins (these are PiTFT defaults):
cs_pin = digitalio.DigitalInOut(board.CE0)
dc_pin = digitalio.DigitalInOut(board.D25)
reset_pin = digitalio.DigitalInOut(board.D24)

# Config for display baudrate (default max is 24mhz):
```

```

BAUDRATE = 24000000

# Setup SPI bus using hardware SPI:
spi = board.SPI()

# pylint: disable=line-too-long
# Create the display:
# disp = st7789.ST7789(spi, rotation=90, # 2.0" ST7789
# disp = st7789.ST7789(spi, height=240, y_offset=80, rotation=180, # 1.3", 1.54"
ST7789
# disp = st7789.ST7789(spi, rotation=90, width=135, height=240, x_offset=53,
y_offset=40, # 1.14" ST7789
# disp = st7789.ST7789(spi, rotation=90, width=172, height=320, x_offset=34, #
1.47" ST7789
# disp = st7789.ST7789(spi, rotation=270, width=170, height=320, x_offset=35, #
1.9" ST7789
# disp = hx8357.HX8357(spi, rotation=180, # 3.5" HX8357
# disp = st7735.ST7735R(spi, rotation=90, # 1.8" ST7735R
# disp = st7735.ST7735R(spi, rotation=270, height=128, x_offset=2, y_offset=3, #
1.44" ST7735R
# disp = st7735.ST7735R(spi, rotation=90, bgr=True, width=80, # 0.96" MiniTFT
Rev A ST7735R
# disp = st7735.ST7735R(spi, rotation=90, invert=True, width=80, # 0.96" MiniTFT
Rev B ST7735R
# x_offset=26, y_offset=1,
# disp = ssd1351.SSD1351(spi, rotation=180, # 1.5" SSD1351
# disp = ssd1351.SSD1351(spi, height=96, y_offset=32, rotation=180, # 1.27" SSD1351
# disp = ssd1331.SSD1331(spi, rotation=180, # 0.96" SSD1331
disp = ili9341.ILI9341(
    spi,
    rotation=90, # 2.2", 2.4", 2.8", 3.2" ILI9341
    cs=cs_pin,
    dc=dc_pin,
    rst=reset_pin,
    baudrate=BAUDRATE,
)
# pylint: enable=line-too-long

# Create blank image for drawing.
# Make sure to create image with mode 'RGB' for full color.
if disp.rotation % 180 == 90:
    height = disp.width # we swap height/width to rotate it to landscape!
    width = disp.height
else:
    width = disp.width # we swap height/width to rotate it to landscape!
    height = disp.height

image = Image.new("RGB", (width, height))

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

# Draw a green filled box as the background
draw.rectangle((0, 0, width, height), fill=(0, 255, 0))
disp.image(image)

# Draw a smaller inner purple rectangle
draw.rectangle(
    (BORDER, BORDER, width - BORDER - 1, height - BORDER - 1), fill=(170, 0, 136)
)

# Load a TTF Font
font = ImageFont.truetype("/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf",
    FONTSIZE)

# Draw Some Text
text = "Hello World!"
(font_width, font_height) = font.getsize(text)
draw.text(

```

```

        (width // 2 - font_width // 2, height // 2 - font_height // 2),
        text,
        font=font,
        fill=(255, 255, 0),
    )

# Display image.
disp.image(image)

```

Just like in the last example, we'll do our imports, but this time we're including the **ImageFont** Pillow module because we'll be drawing some text this time.

```

import digitalio
import board
from PIL import Image, ImageDraw, ImageFont
import adafruit_rgb_display.ili9341 as ili9341

```

Next we'll define some parameters that we can tweak for various displays. The **BORDER** will be the size in pixels of the green border between the edge of the display and the inner purple rectangle. The **FONTSIZE** will be the size of the font in points so that we can adjust it easily for different displays.

```

BORDER = 20
FONTSIZE = 24

```

Next, just like in the previous example, we will set up the display, setup the rotation, and create a draw object. If you have are using a different display than the ILI9341, go ahead and adjust your initializer as explained in the previous example. After that, we will setup the background with a green rectangle that takes up the full screen. To get green, we pass in a tuple that has our Red, Green, and Blue color values in it in that order which can be any integer from **0** to **255**.

```

draw.rectangle((0, 0, width, height), fill=(0, 255, 0))
disp.image(image)

```

Next we will draw an inner purple rectangle. This is the same color value as our example in displayio quickstart, except the hexadecimal values have been converted to decimal. We use the **BORDER** parameter to calculate the size and position that we want to draw the rectangle.

```

draw.rectangle((BORDER, BORDER, width - BORDER - 1, height - BORDER - 1),
               fill=(170, 0, 136))

```

Next we'll load a TTF font. The **DejaVuSans.ttf** font should come preloaded on your Pi in the location in the code. We also make use of the **FONTSIZE** parameter that we discussed earlier.


```
# Load a TTF Font
font = ImageFont.truetype('/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf',
    FONTSIZE)
```

Now we draw the text Hello World onto the center of the display. You may recognize the centering calculation was the same one we used to center crop the image in the previous example. In this example though, we get the font size values using the `getsize()` function of the font object.

```
# Draw Some Text
text = "Hello World!"
(font_width, font_height) = font.getsize(text)
draw.text((width//2 - font_width//2, height//2 - font_height//2),
    text, font=font, fill=(255, 255, 0))
```

Finally, just like before, we display the image.

```
disp.image(image)
```



Displaying System Information

In this last example we'll take a look at getting the system information and displaying it. This can be very handy for system monitoring. Here's the code for that example:

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

"""
This will show some Linux Statistics on the attached display. Be sure to adjust
to the display you have connected. Be sure to check the learn guides for more
usage information.
```

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 time
import subprocess
import digitalio
import board
from PIL import Image, ImageDraw, ImageFont
from adafruit_rgb_display import ili9341
from adafruit_rgb_display import st7789 # pylint: disable=unused-import
from adafruit_rgb_display import hx8357 # pylint: disable=unused-import
from adafruit_rgb_display import st7735 # pylint: disable=unused-import
from adafruit_rgb_display import ssd1351 # pylint: disable=unused-import
from adafruit_rgb_display import ssd1331 # pylint: disable=unused-import

# Configuration for CS and DC pins (these are PiTFT defaults):
cs_pin = digitalio.DigitalInOut(board.CE0)
dc_pin = digitalio.DigitalInOut(board.D25)
reset_pin = digitalio.DigitalInOut(board.D24)

# Config for display baudrate (default max is 24mhz):
BAUDRATE = 24000000

# Setup SPI bus using hardware SPI:
spi = board.SPI()

# pylint: disable=line-too-long
# Create the display:
# disp = st7789.ST7789(spi, rotation=90, # 2.0" ST7789
# disp = st7789.ST7789(spi, height=240, y_offset=80, rotation=180, # 1.3", 1.54"
ST7789
# disp = st7789.ST7789(spi, rotation=90, width=135, height=240, x_offset=53,
y_offset=40, # 1.14" ST7789
# disp = st7789.ST7789(spi, rotation=90, width=172, height=320, x_offset=34, #
1.47" ST7789
# disp = st7789.ST7789(spi, rotation=270, width=170, height=320, x_offset=35, #
1.9" ST7789
# disp = hx8357.HX8357(spi, rotation=180, # 3.5" HX8357
# disp = st7735.ST7735R(spi, rotation=90, # 1.8" ST7735R
# disp = st7735.ST7735R(spi, rotation=270, height=128, x_offset=2, y_offset=3, #
1.44" ST7735R
# disp = st7735.ST7735R(spi, rotation=90, bgr=True, width=80, # 0.96" MiniTFT
Rev A ST7735R
# disp = st7735.ST7735R(spi, rotation=90, invert=True, width=80, # 0.96" MiniTFT
Rev B ST7735R
# x_offset=26, y_offset=1,
# disp = ssd1351.SSD1351(spi, rotation=180, # 1.5" SSD1351
# disp = ssd1351.SSD1351(spi, height=96, y_offset=32, rotation=180, # 1.27" SSD1351
# disp = ssd1331.SSD1331(spi, rotation=180, # 0.96" SSD1331
disp = ili9341.ILI9341(
    spi,
    rotation=90, # 2.2", 2.4", 2.8", 3.2" ILI9341
    cs=cs_pin,
    dc=dc_pin,
    rst=reset_pin,
    baudrate=BAUDRATE,
)
# pylint: enable=line-too-long

# Create blank image for drawing.
# Make sure to create image with mode 'RGB' for full color.
if disp.rotation % 180 == 90:
    height = disp.width # we swap height/width to rotate it to landscape!
    width = disp.height
else:
    width = disp.width # we swap height/width to rotate it to landscape!
```

```

    height = disp.height

image = Image.new("RGB", (width, height))

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

# Draw a black filled box to clear the image.
draw.rectangle((0, 0, width, height), outline=0, fill=(0, 0, 0))
disp.image(image)

# First define some constants to allow easy positioning of text.
padding = -2
x = 0

# Load a TTF font. Make sure the .ttf font file is in the
# same directory as the python script!
# Some other nice fonts to try: http://www.dafont.com/bitmap.php
font = ImageFont.truetype("/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf", 24)

while True:
    # Draw a black filled box to clear the image.
    draw.rectangle((0, 0, width, height), outline=0, fill=0)

    # Shell scripts for system monitoring from here:
    # https://unix.stackexchange.com/questions/119126/command-to-display-memory-
usage-disk-usage-and-cpu-load
    cmd = "hostname -I | cut -d' ' -f1"
    IP = "IP: " + subprocess.check_output(cmd, shell=True).decode("utf-8")
    cmd = "top -bn1 | grep load | awk '{printf \"CPU Load: %.2f\\\", $(NF-2)}'"
    CPU = subprocess.check_output(cmd, shell=True).decode("utf-8")
    cmd = "free -m | awk 'NR==2{printf \"Mem: %s/%s MB %.2f%\\\",
$3,$2,$3*100/$2 }'"
    MemUsage = subprocess.check_output(cmd, shell=True).decode("utf-8")
    cmd = 'df -h | awk \'$NF=="\/"{printf "Disk: %d/%d GB %s", $3,$2,$5}\''
    Disk = subprocess.check_output(cmd, shell=True).decode("utf-8")
    cmd = "cat /sys/class/thermal/thermal_zone0/temp | awk '{printf \"CPU Temp: %.
1f C\\\", $(NF-0) / 1000}'" # pylint: disable=line-too-long
    Temp = subprocess.check_output(cmd, shell=True).decode("utf-8")

    # Write four lines of text.
    y = padding
    draw.text((x, y), IP, font=font, fill="#FFFFFF")
    y += font.getsize(IP)[1]
    draw.text((x, y), CPU, font=font, fill="#FFFF00")
    y += font.getsize(CPU)[1]
    draw.text((x, y), MemUsage, font=font, fill="#00FF00")
    y += font.getsize(MemUsage)[1]
    draw.text((x, y), Disk, font=font, fill="#0000FF")
    y += font.getsize(Disk)[1]
    draw.text((x, y), Temp, font=font, fill="#FF00FF")

    # Display image.
    disp.image(image)
    time.sleep(0.1)

```

Just like the last example, we'll start by importing everything we imported, but we're adding two more imports. The first one is `time` so that we can add a small delay and the other is `subprocess` so we can gather some system information.

```

import time
import subprocess
import digitalio
import board

```

```
from PIL import Image, ImageDraw, ImageFont
import adafruit_rgb_display.ili9341 as ili9341
```

Next, just like in the first two examples, we will set up the display, setup the rotation, and create a draw object. If you have are using a different display than the ILI9341, go ahead and adjust your initializer as explained in the previous example.

Just like in the first example, we're going to draw a black rectangle to fill up the screen. After that, we're going to set up a couple of constants to help with positioning text. The first is the `padding` and that will be the Y-position of the top-most text and the other is `x` which is the X-Position and represents the left side of the text.

```
# First define some constants to allow easy positioning of text.
padding = -2
x = 0
```

Next, we load a font just like in the second example.

```
font = ImageFont.truetype('/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf', 24)
```

Now we get to the main loop and by using `while True:`, it will loop until Control+C is pressed on the keyboard. The first item inside here, we clear the screen, but notice that instead of giving it a tuple like before, we can just pass `0` and it will draw black.

```
draw.rectangle((0, 0, width, height), outline=0, fill=0)
```

Next, we run a few scripts using the `subprocess` function that get called to the Operating System to get information. The in each command is passed through `awk` in order to be formatted better for the display. By having the OS do the work, we don't have to. These little scripts came from <https://unix.stackexchange.com/questions/119126/command-to-display-memory-usage-disk-usage-and-cpu-load>

```
cmd = "hostname -I | cut -d\ ' ' -f1"
IP = "IP: "+subprocess.check_output(cmd, shell=True).decode("utf-8")
cmd = "top -bn1 | grep load | awk '{printf \"CPU Load: %.2f\\\", $(NF-2)}'"
CPU = subprocess.check_output(cmd, shell=True).decode("utf-8")
cmd = "free -m | awk 'NR==2{printf \"Mem: %s/%s MB %.2f%%\\\", $3,$2,$3*100/$2 }'"
MemUsage = subprocess.check_output(cmd, shell=True).decode("utf-8")
cmd = "df -h | awk '$NF==\"/\	\"{printf \"Disk: %d/%d GB %s\\\", $3,$2,$5}'"
Disk = subprocess.check_output(cmd, shell=True).decode("utf-8")
cmd = "cat /sys/class/thermal/thermal_zone0/temp | awk \'{printf \"CPU Temp: %.1f C\\\", $(NF-0) / 1000}\	'" # pylint: disable=line-too-long
Temp = subprocess.check_output(cmd, shell=True).decode("utf-8")
```

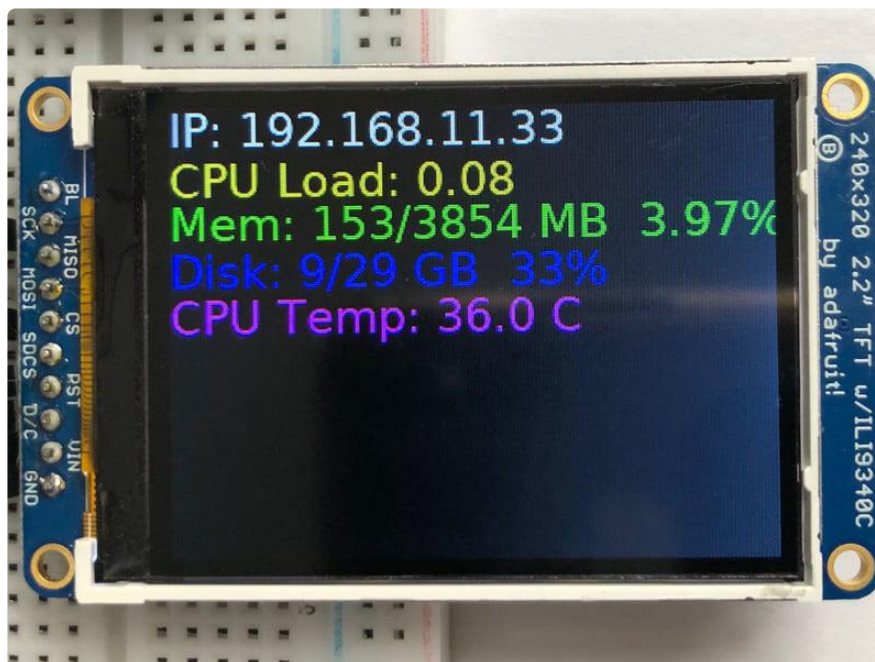
Now we display the information for the user. Here we use yet another way to pass color information. We can pass it as a color string using the pound symbol, just like we

would with HTML. With each line, we take the height of the line using `getsize()` and move the pointer down by that much.

```
y = padding
draw.text((x, y), IP, font=font, fill="#FFFFFF")
y += font.getsize(IP)[1]
draw.text((x, y), CPU, font=font, fill="#FFFF00")
y += font.getsize(CPU)[1]
draw.text((x, y), MemUsage, font=font, fill="#00FF00")
y += font.getsize(MemUsage)[1]
draw.text((x, y), Disk, font=font, fill="#0000FF")
y += font.getsize(Disk)[1]
draw.text((x, y), Temp, font=font, fill="#FF00FF")
```

Finally, we write all the information out to the display using `disp.image()`. Since we are looping, we tell Python to sleep for `0.1` seconds so that the CPU never gets too busy.

```
disp.image(image)
time.sleep(.1)
```



Downloads and Links

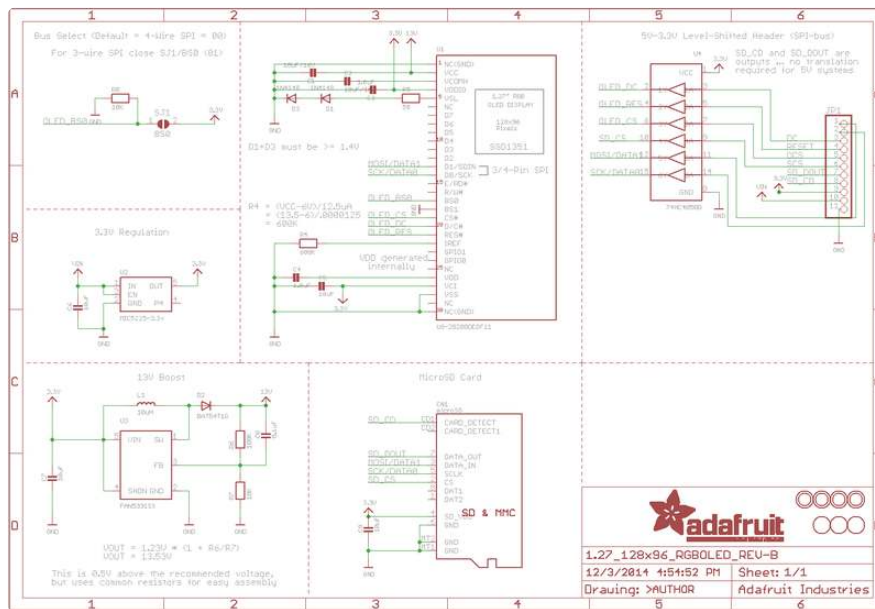
Data Sheets:

- [SSD1351 Display Controller Datasheet \(\)](#)
- [1.5" OLED Display Module datasheet \(\)](#)
- [Fritzing objects in the Adafruit Fritzing library \(\)](#)
- [EagleCAD PCB for 1.27" Color OLED \(\)](#)

- [EagleCAD PCB for the 1.5" Color OLED \(\)](#)

Schematic

Click to enlarge



For the level shifter we use the [CD74HC4050 \(\)](#) which has a typical propagation delay of ~10ns