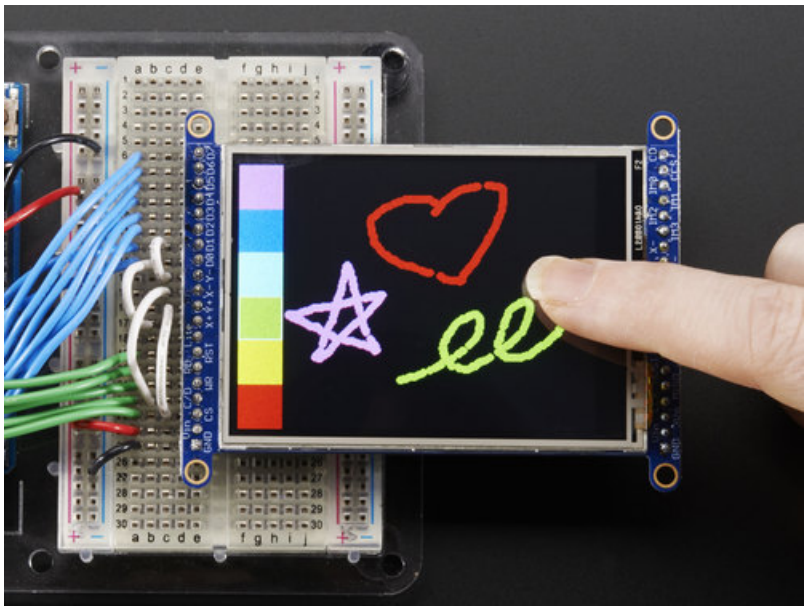
adafruit learning system

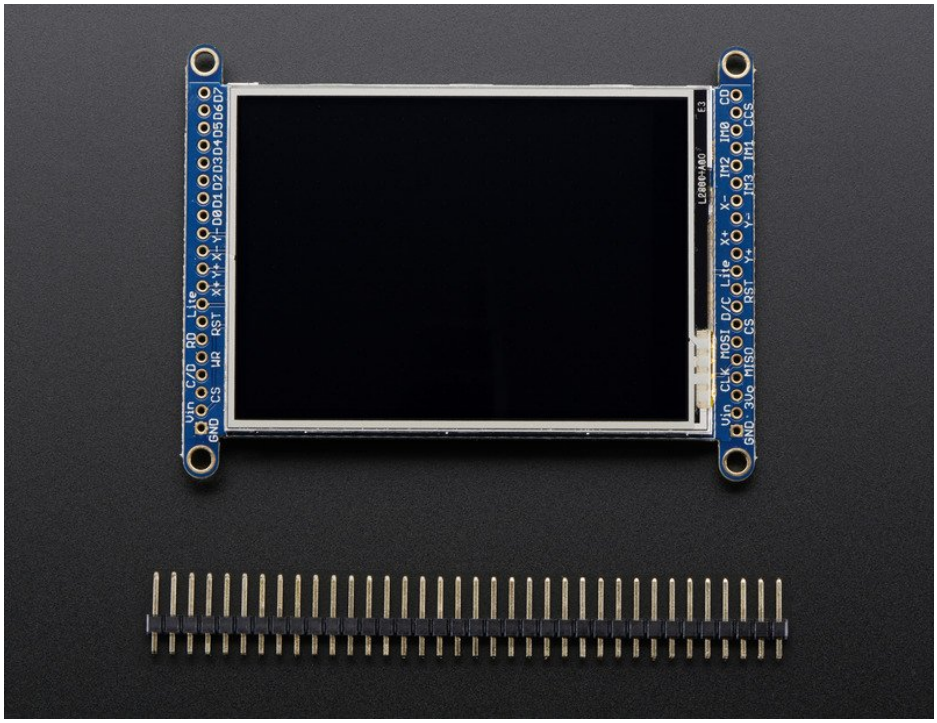
Adafruit 2.8" and 3.2" Color TFT Touchscreen Breakout v2

Created by lady ada



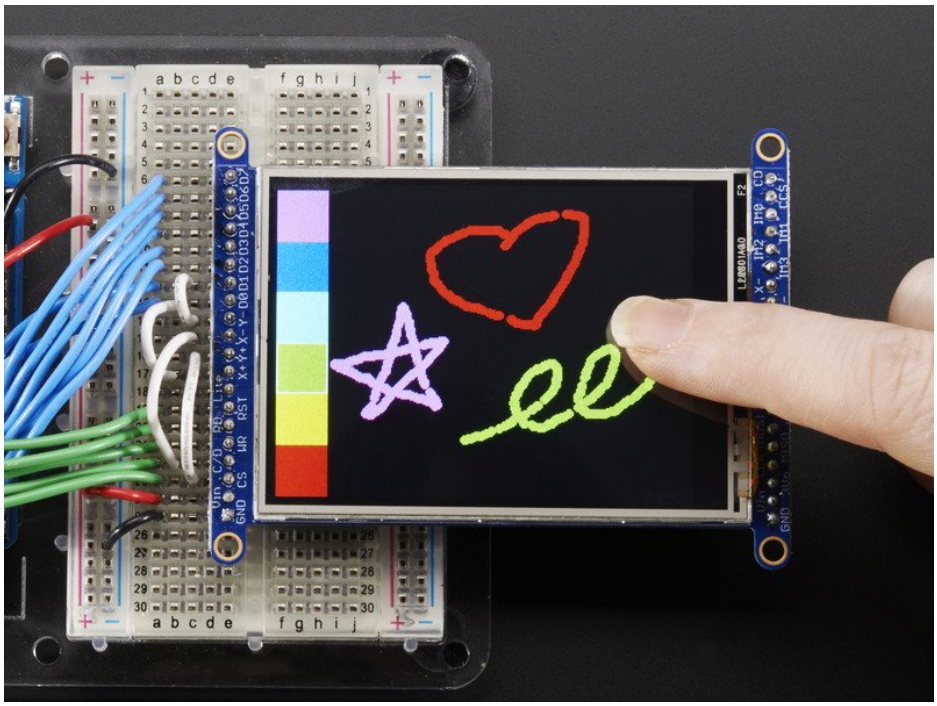
Last updated on 2020-06-19 03:59:40 PM EDT

Overview



Add some jazz & pizazz to your project with a color touchscreen LCD. These TFT displays are big (2.8" or 3.2" diagonal) bright (4 or 6 white-LED backlight) and colorful! 240x320 pixels with individual RGB pixel control, this has way more resolution than a black and white 128x64 display.

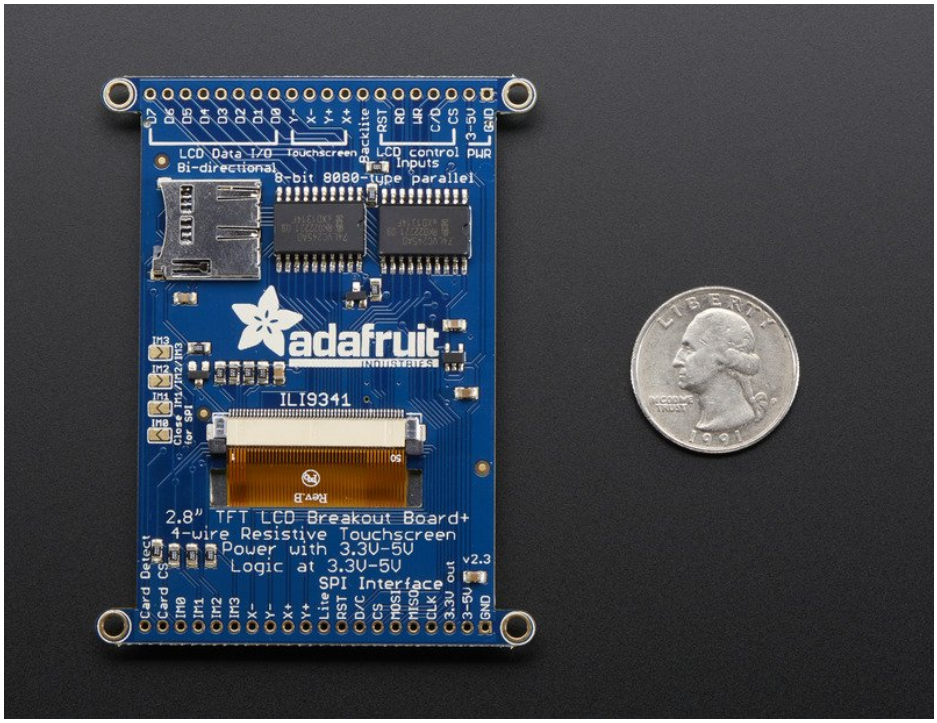
As a bonus, this display has either a resistive or capacitive touchscreen attached to it already, so you can detect finger presses anywhere on the screen.



This display has a controller built into it with RAM buffering, so that almost no work is done by the microcontroller. **The display can be used in two modes: 8-bit or SPI.** For 8-bit mode, you'll need 8 digital data lines and 4 or 5 digital control lines to read and write to the display (12 lines total). SPI mode requires only 5 pins total (SPI data in, data out, clock, select, and d/c) but is slower than 8-bit mode.

If you have the **resistive** touch version, 4 pins are required for the touch screen (2 digital, 2 analog) or you can purchase and use our resistive touchscreen controller (not included) to use I2C or SPI (http://adafru.it/1571)

If you have the **capacitive** touch version, there is a capacitive touch controller chip already installed that communicates of standard I2C plus an IRQ line.

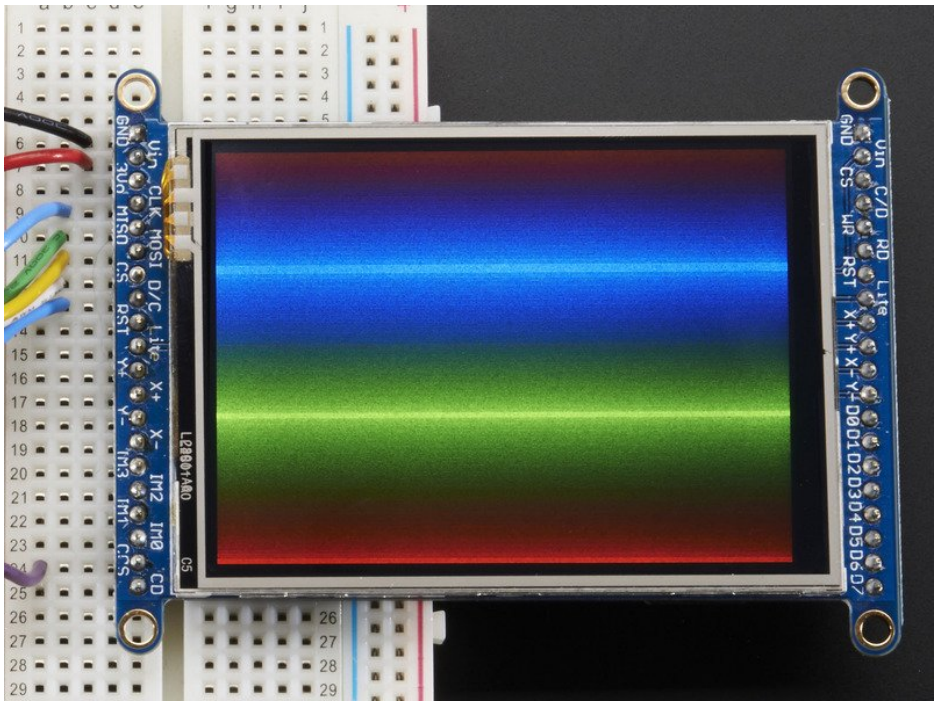


Of course, we wouldn't just leave you with a datasheet and a "good luck!". For 8-bit interface fans we've written a full open source graphics library that can draw pixels, lines, rectangles, circles, text, and more (https://adafru.it/aHk). For SPI users, we have a library as well (https://adafru.it/d4d), its separate from the 8-bit library since both versions are heavily optimized.

For resitive touch, we also have a touch screen library that detects x, y and z (pressure) (https://adafru.it/aT1) and example code to demonstrate all of it.

For capacitive touch, we have an I2C interface library for the captouch chip. (https://adafru.it/dGG)

If you are using an Arduino-shaped microcontroller, check out our TFT shield version of this same display, with SPI control and a touch screen controller as well (http://adafru.it/1651)

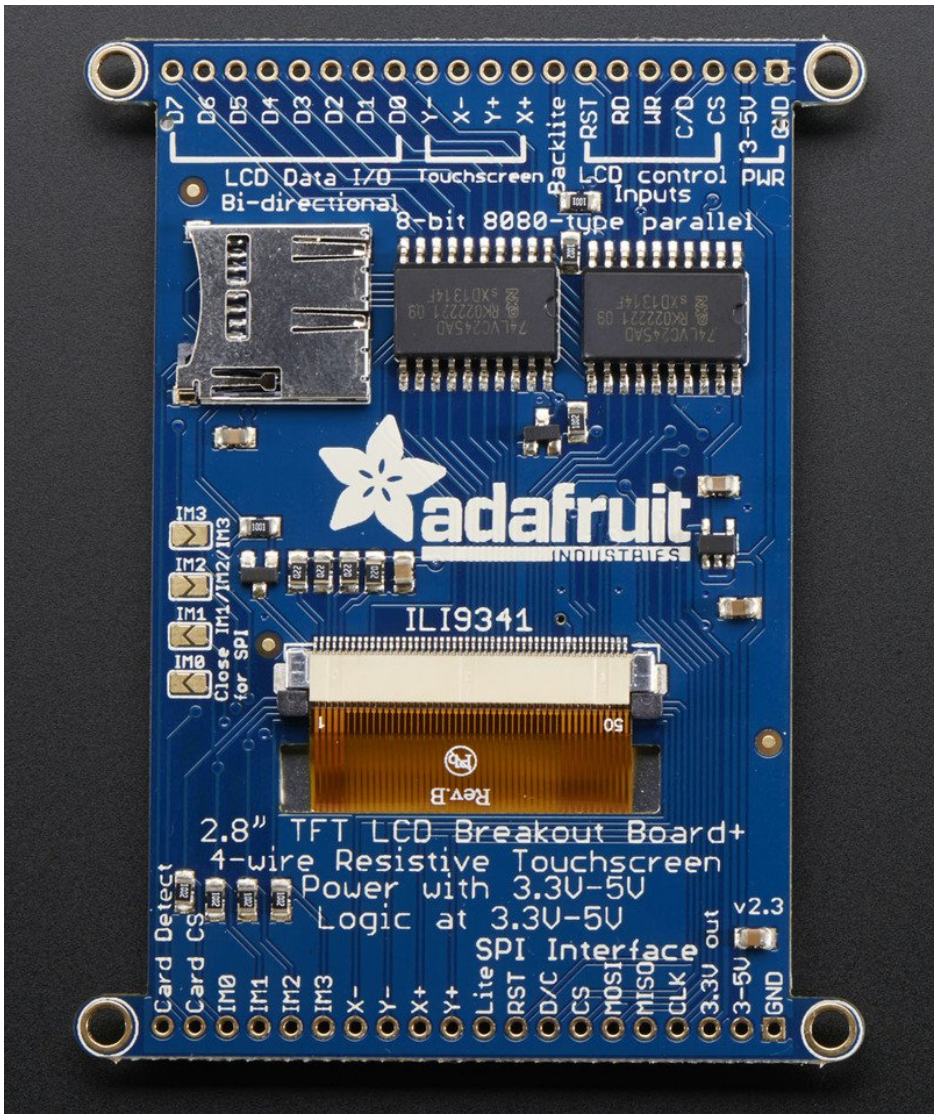

GND
Vin
3Vo
CLK
MOSI
D/C
Lite
MISO
CS
RST
Y+
X+
Y-
X-
IM3
IM2
IM1
IM0
CD
CS
C/D
WR
RD
RST
Lite
X+
Y+
X-
Y-
D0
D1
D2
D3
D4
D5
D6
D7
LC8001A0
C5

Pinouts

The 2.8" and 3.2" TFT display on this breakout supports many different modes - so many that the display itself has 50 pins. However, we think most people really only use 2 different modes, either "SPI" mode or 8-bit mode (which includes both 6800 and 8080). Each 'side' of the display has all the pins required for that mode. You can switch between modes, by rewiring the display, but it cannot be used in two modes at the same time!

All logic pins, both 8-bit and SPI sides, are 3-5V logic level compatible, the 74LVX245 chips on the back perform fast level shifting so you can use either kind of logic levels. If there's data output, the levels are at at 3.3V

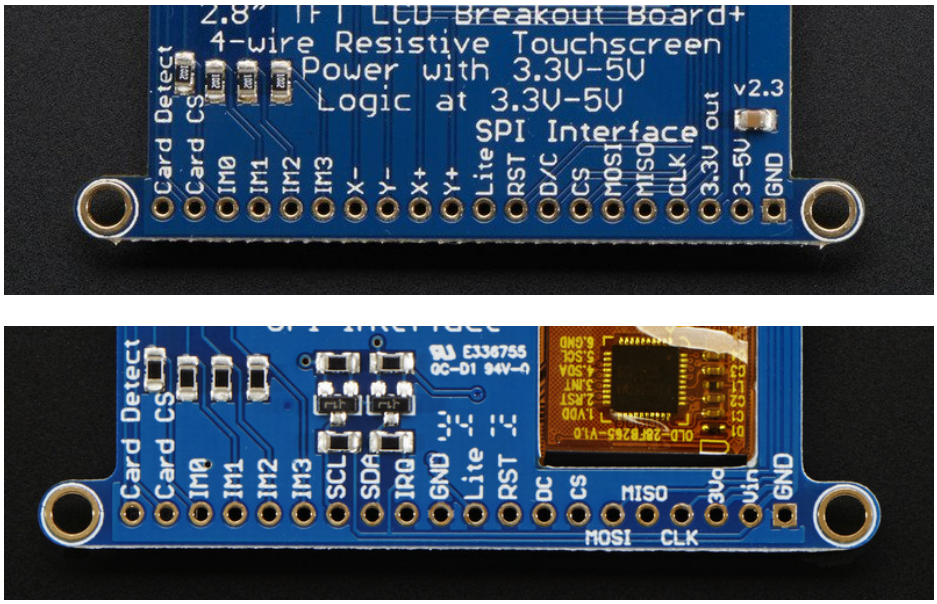
We show the 2.8" version of this breakout in the photos below but the 3.2" TFT is identical, just a lil bit bigger



SPI Mode

This is what we think will be a popular mode when speed is not of the utmost importance. It doesn't use as many pins (only 4 to draw on the TFT if you skip the MISO pin), is fairly flexible, and easy to port to various microcontrollers. It also allows using a microSD card socket on the same SPI bus. However, its slower than parallel 8-bit mode because you

have to send each bit at a time instead of 8-bits at a time. Tradeoffs!



- **GND** - this is the power and signal ground pin
- **3-5V / Vin** - this is the power pin, connect to 3-5VDC - it has reverse polarity protection but try to wire it right!
- **3.3Vout** - this is the 3.3V output from the onboard regulator
- **CLK** - this is the SPI clock input pin
- **MISO** - this is the SPI Microcontroller In Serial Out pin, its used for the SD card mostly, and for debugging the TFT display. It isn't necessary for using the TFT display which is write-only
- **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 TFT
- **CS** - this is the TFT SPI chip select pin
- **D/C** - this is the TFT SPI data or command selector pin
- **RST** - this is the TFT reset pin. There's auto-reset circuitry on the breakout so this pin is not required but it can be helpful sometimes to reset the TFT if your setup is not always resetting cleanly. Connect to ground to reset the TFT
- **Lite** - this is the PWM input for the backlight control. It is by default pulled high (backlight on) you can PWM at any frequency or pull down to turn the backlight off
- **IM3 IM2 IM1 IM0** - these are interface control set pins. In general these breakouts aren't used, and instead the onboard jumpers are used to fix the interface to SPI or 8-bit. However, we break these out for advanced use and also for our test procedures
- **Card CS / CCS** - this is the SD card chip select, used if you want to read from the SD card.
- **Card Detect / CD** - this is the SD card detect pin, it floats when a card is inserted, and tied to ground when the card is not inserted. We don't use this in our code but you can use this as a switch to detect if an SD card is in place without trying to electrically query it. Don't forget to use a pullup on this pin if so!

Resistive touch pins

- **Y+ X+ Y- X-** these are the 4 resistive touch screen pads, which can be read with analog pins to determine touch points. They are completely separated from the TFT electrically (the overlay is glued on top)

Capacitive touch pins

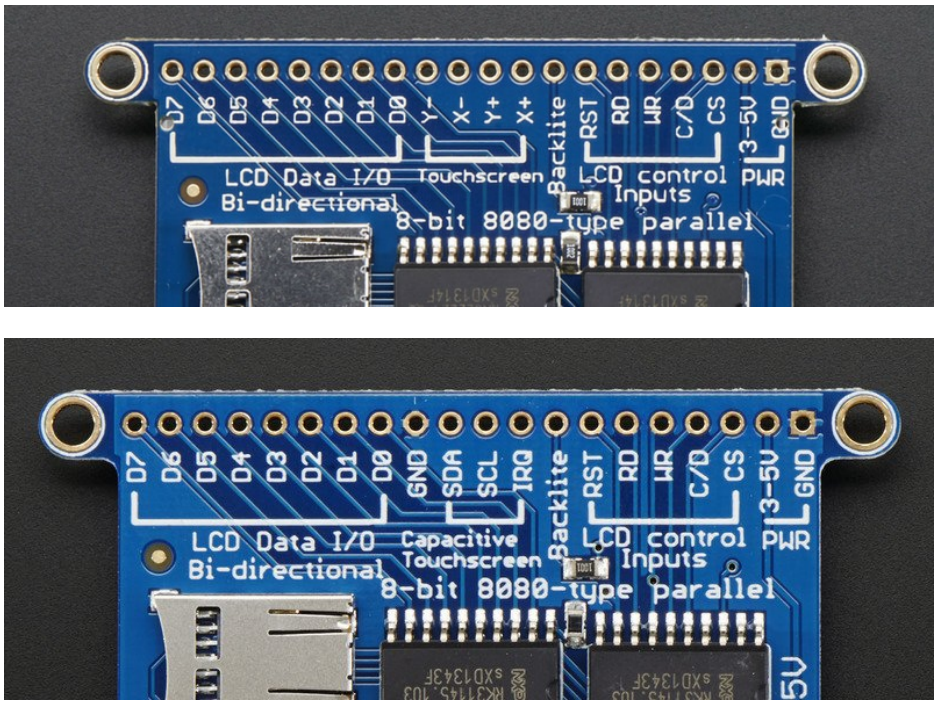
- **SDA** - this is the I2C data pin for the captouch chip, there's level shifting on this pin so you can use 3-5V logic.

There's also a 10K pullup

- **SCL** - this is the I2C clock pin for the captouch chip, there's level shifting on this pin so you can use 3-5V logic. There's also a 10K pullup
- **IRQ** - this is the captouch interrupt pin. When a touch is detected, this pin goes low.

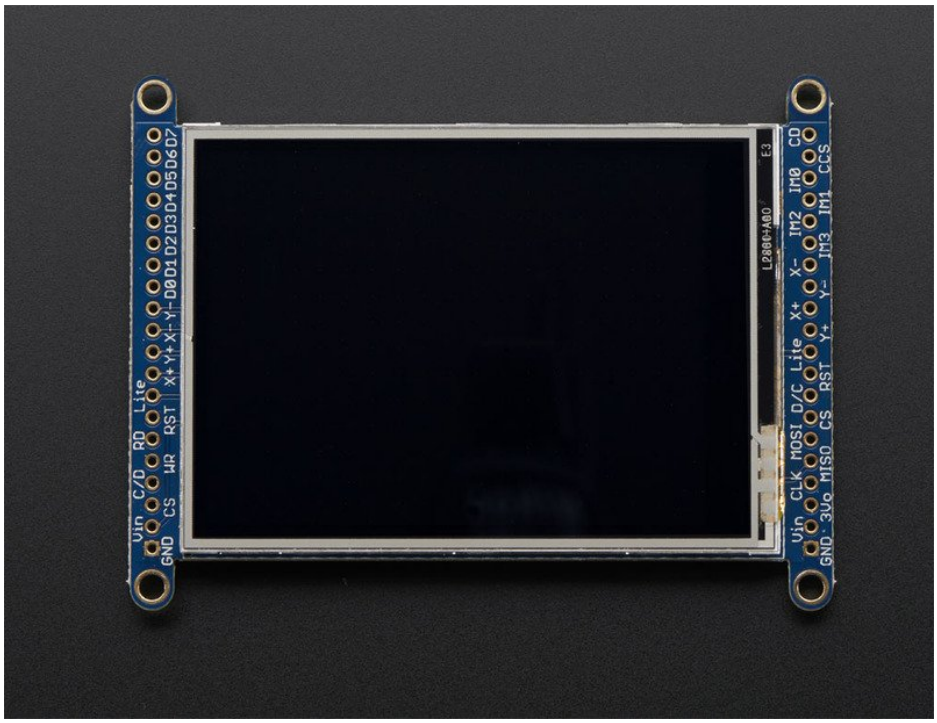
8-Bit Mode

This mode is for when you have lots of pins and want more speed. In this mode we send 8 bits at a time, so it needs way more pins, 12 or so (8 bits plus 4 control)! This isn't recommended because most microcontrollers don't have a ton of pins and also we optimize our libraries for SPI!



- **GND** - this is the power and signal ground pin
- **3-5V (Vin)**- this is the power pin, connect to 3-5VDC - it has reverse polarity protection but try to wire it right!
- **CS** - this is the TFT 8-bit chip select pin (it is also tied to the SPI mode **CS** pin)
- **C/D** - this is the TFT 8-bit data or command selector pin. It is **not the same as the SPI D/C pin!** Instead, it's the same as the SPI CLK pin.
- **WR** - this is the TFT 8-bit write strobe pin. It is also connected to the SPI **D/C** pin
- **RD** - this is the TFT 8-bit read strobe pin. You may not need this pin if you don't want to read data from the display
- **RST** - this is the TFT reset pin. There's auto-reset circuitry on the breakout so this pin is not required but it can be helpful sometimes to reset the TFT if your setup is not always resetting cleanly. Connect to ground to reset the TFT
- **Backkite** - this is the PWM input for the backlight control. It is by default pulled high (backlight on) you can PWM at any frequency or pull down to turn the backlight off
- **D0** thru **D7** - these are the 8 bits of parallel data sent to the TFT in 8-bit mode. **D0** is the least-significant-bit and **D7** is the MSB

Wiring and Test



We tried to make this TFT breakout useful for both high-pin microcontrollers that can handle 8-bit data transfer modes as well as low-pincount micros like the Arduino UNO and Leonardo that are OK with SPI.

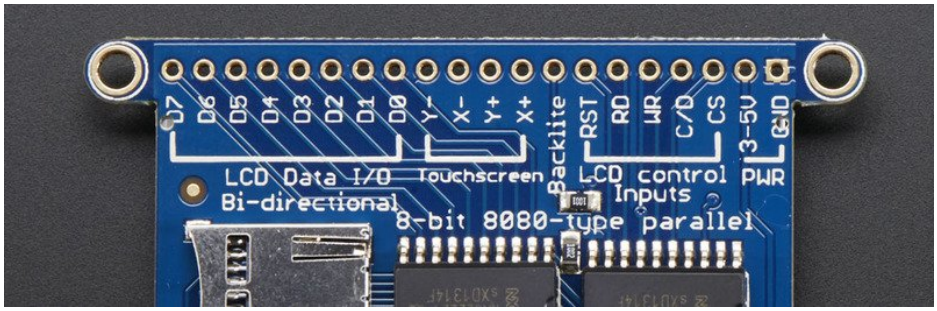
Essentially, the tradeoff is pins for speed. SPI is about 2-4 times slower than 8-bit mode, but that may not matter for basic graphics!

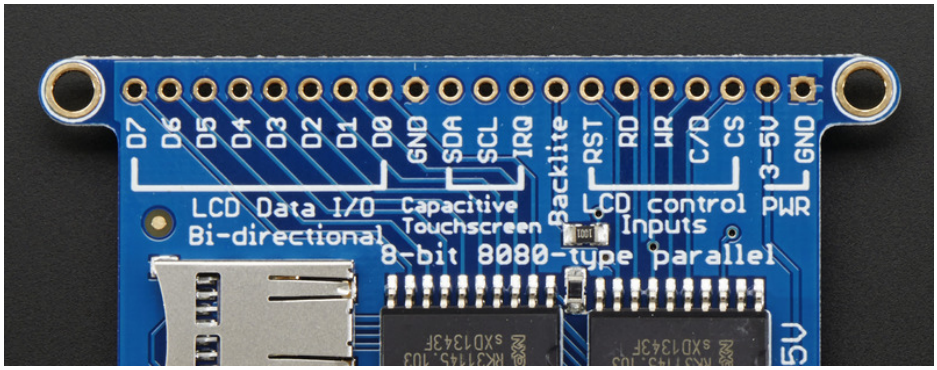
In addition, SPI mode has the benefit of being able to use the onboard microSD card socket for reading images. We don't have support for this in 8-bit mode so if you want to have an all-in-one image viewer type application, use SPI!

8-Bit Wiring

Wiring up the 8-bit mode is kind of a pain, so we really only recommend doing it for UNO (which we show) and Mega (which we describe, and is pretty easy since its 8 pins in a row). Anything else, like a Leonardo or Micro, we strongly recommend going with SPI mode since we don't have an example for that. Any other kind of 'Arduino compatible' that isn't an Uno, try SPI first. The 8-bit mode is hand-tweaked in the **Adafruit_TFTLCD pin_magic.h** file. Its really only for advanced users who are totally cool with figuring out bitmasks for various ports & pins.

Really, we'll show how to do the UNO but anything else? go with SPI!





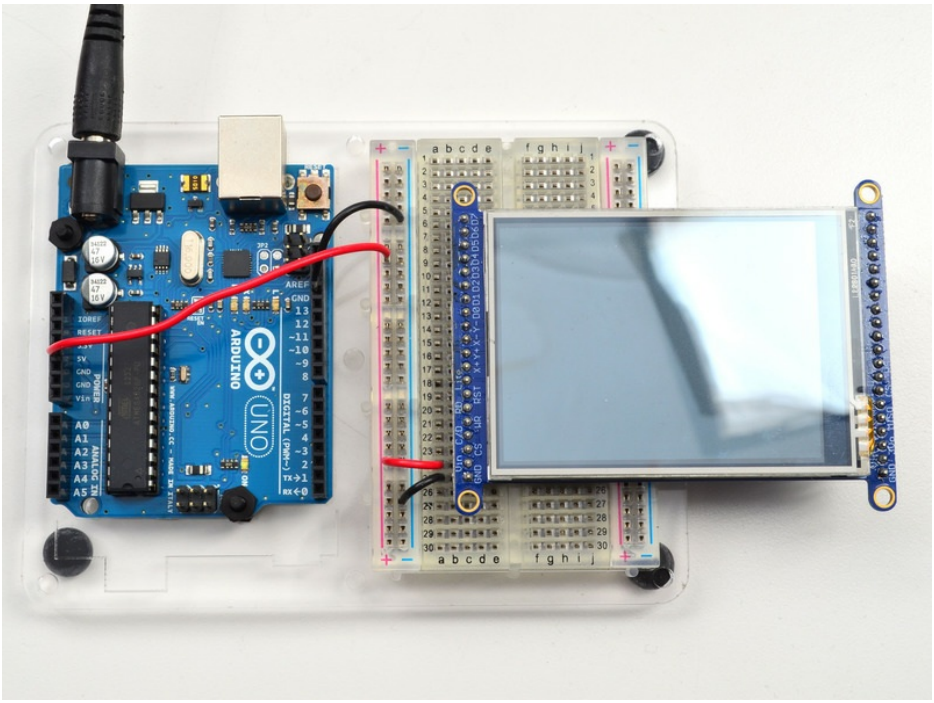
We show the 2.8" version of this breakout in the photos below but the 3.2" TFT is identical, just a lil bit bigger

Make sure you're soldering and connecting to the 8-bit side!

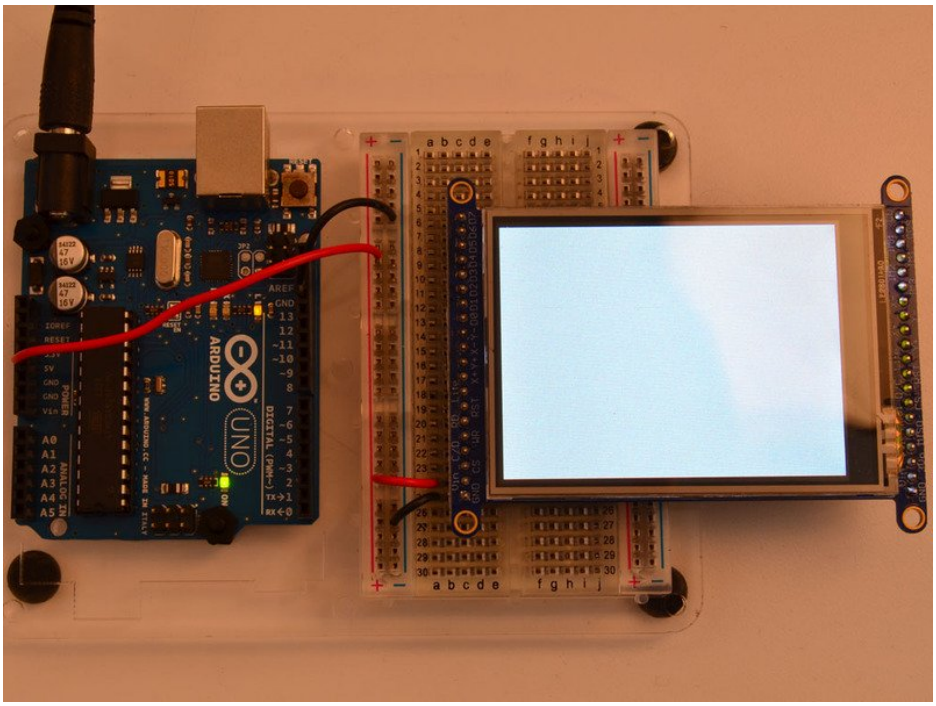
Part 1 - Power & backlight test

Begin by wiring up the **3-5VDC** and **GND** pins.

Connect the **3-5V** pin to **5V** and **GND** to **GND** on your Arduino. I'm using the breadboard rails but you can also just wire directly.

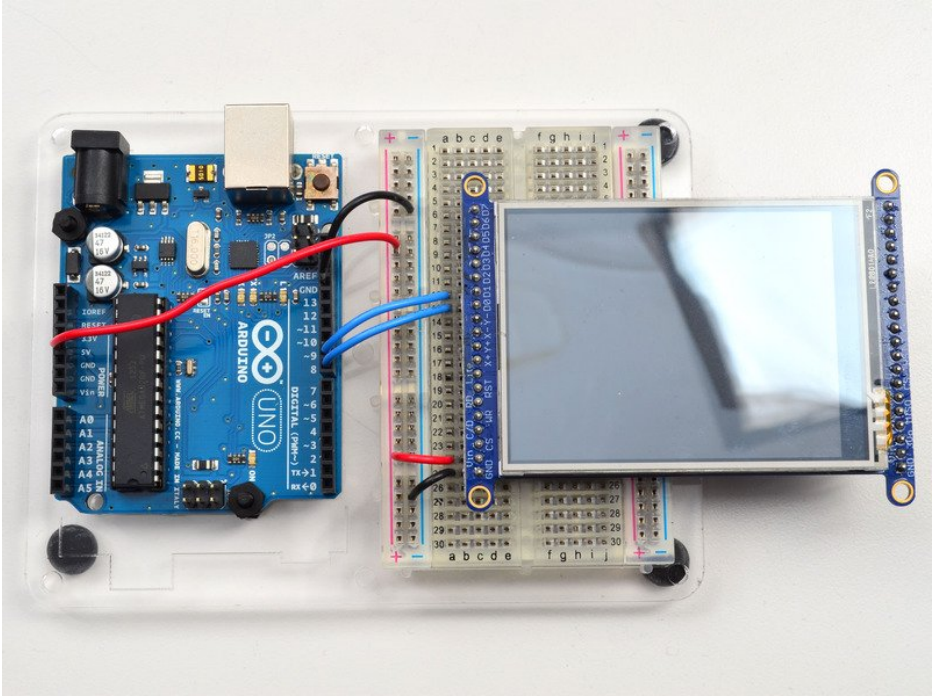


Power it up and you should see the white backlight come on.

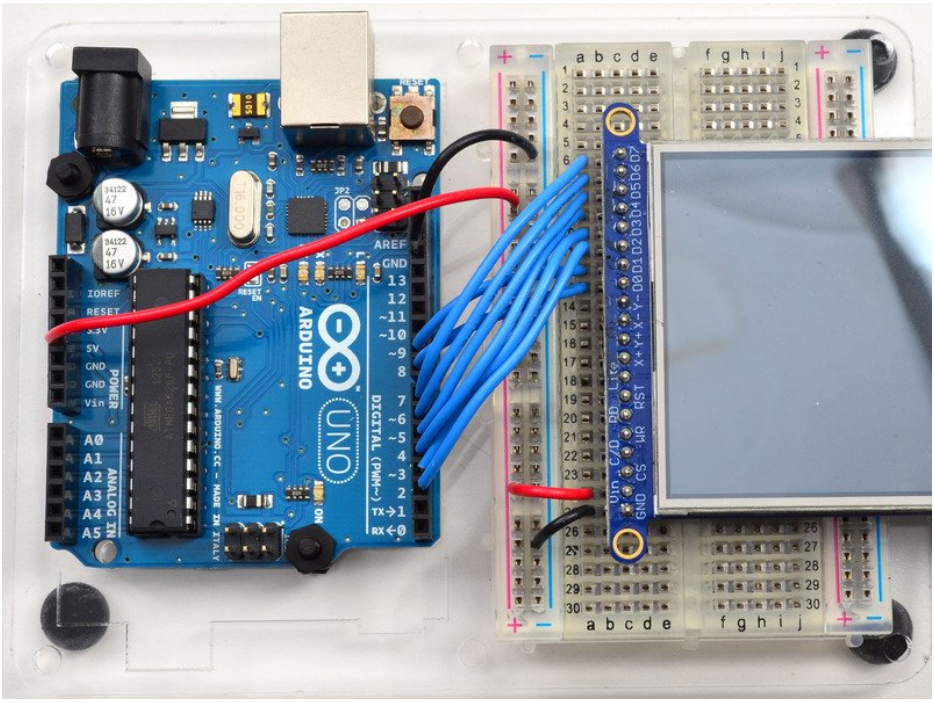


Part 2 - Data Bus Lines

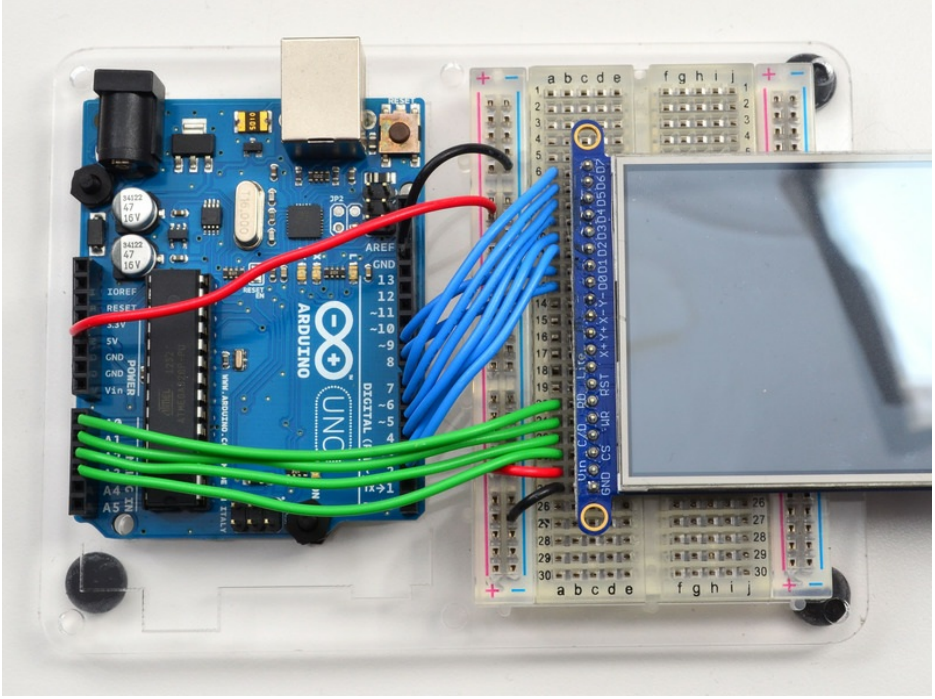
Now that the backlight is working, we can get the TFT LCD working. There are many pins required, and to keep the code running fairly fast, we have 'hardcoded' Arduino digital pins **#2-#9** for the 8 data lines.
However, they are not in that order! D0 and D1 go to **digital #8** and **#9**, then D2-D7 connect to **#2** thru **#7**. This is because Arduino pins #0 and #1 are used for serial data so we can't use them



Begin by connecting **D0** and **D1** to digital **#8** and **9** respectively as seen above. If you're using a Mega, connect the TFT Data Pins **D0-D1** to Mega pins **#22-23**, in that order. Those Mega pins are on the 'double' header.

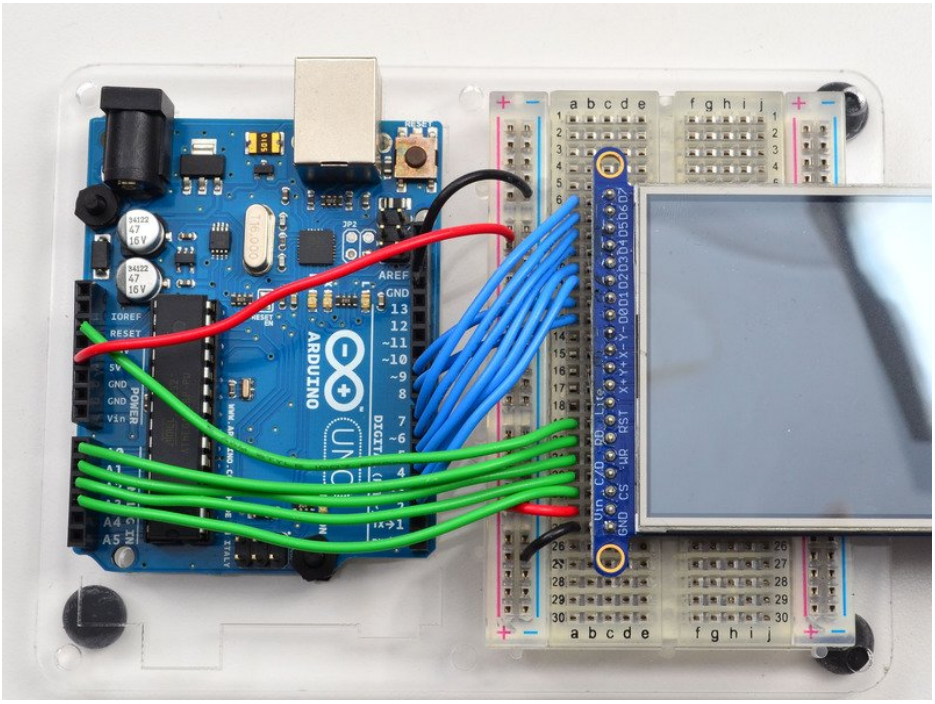


Now you can connect the remaining 6 pins over. Connect **D2-D7** on the TFT pins to digital **2** thru **7** in that order. If you're using a Mega, connect the TFT Data Pins **D2-D7** to Mega pins **#24-29**, in that order. Those Mega pins are on the 'double' header.



In addition to the 8 data lines, you'll also need 4 or 5 control lines. These can later be reassigned to any digital pins, they're just what we have in the tutorial by default.

- Connect the third pin **CS** (**Chip Select**) to Analog 3
- Connect the fourth pin **C/D** (**Command/Data**) to Analog 2
- Connect the fifth pin **WR** (**Write**) to Analog 1
- Connect the sixth pin **RD** (**Read**) to Analog 0



You can connect the seventh pin **RST** (**Reset**) to the Arduino Reset line if you'd like. This will reset the panel when the Arduino is Reset. You can also use a digital pin for the LCD reset if you want to manually reset. There's auto-reset

circuitry on the board so you probably don't need to use this pin at all and leave it disconnected

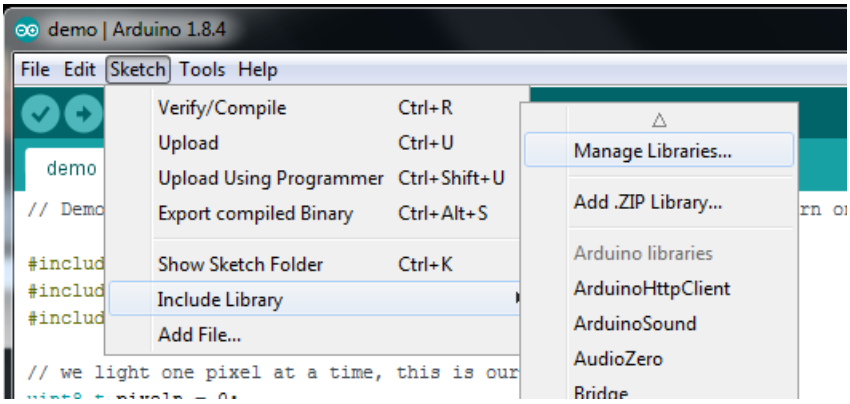
The **RD** pin is used to read the chip ID off the TFT. Later, once you get it all working, you can remove this pin and the ID test, although we suggest keeping it since its useful for debugging your wiring.

OK! Now we can run some code

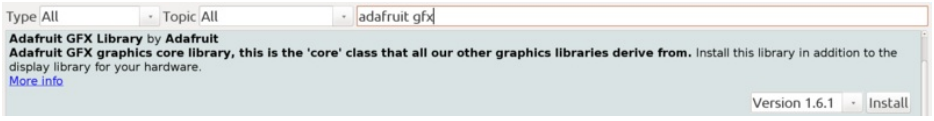
8-Bit Library Install

We have example code ready to go for use with these TFTs. It's written for Arduino, which should be portable to any microcontroller by adapting the C++ source.

Two libraries need to be downloaded and installed: the TFTLCD library (https://adafru.it/aHk) and the GFX library. (https://adafru.it/aJa) You can install these libraries through the Arduino library manager.

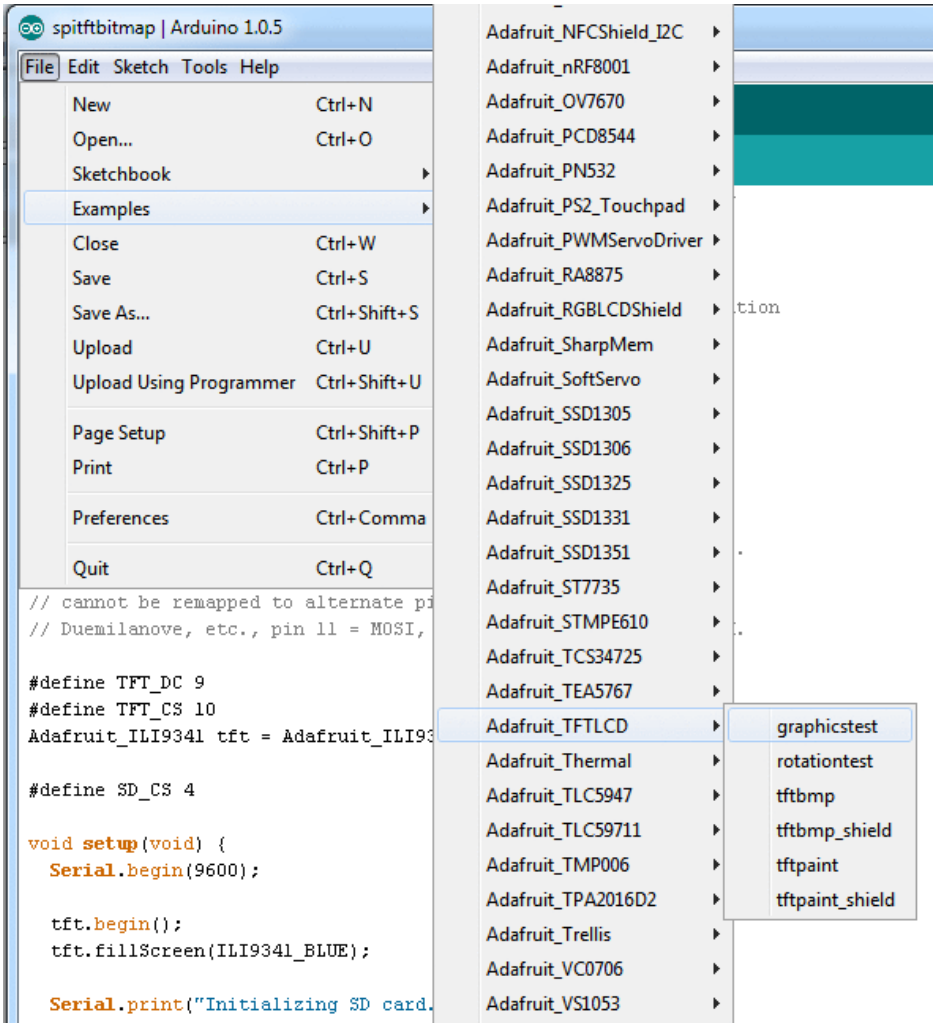


Search for the **Adafruit_GFX** library and install it. If using an older Arduino IDE (pre-1.8.10), also locate and install **Adafruit_BusIO**.



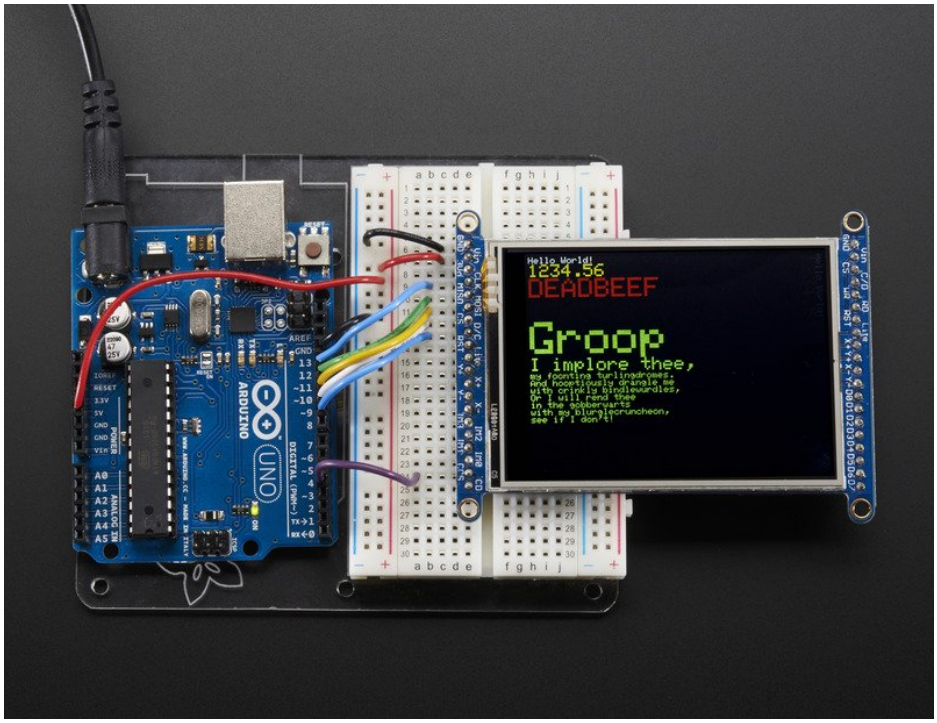
Search for the **Adafruit TFTLCD** library and install it

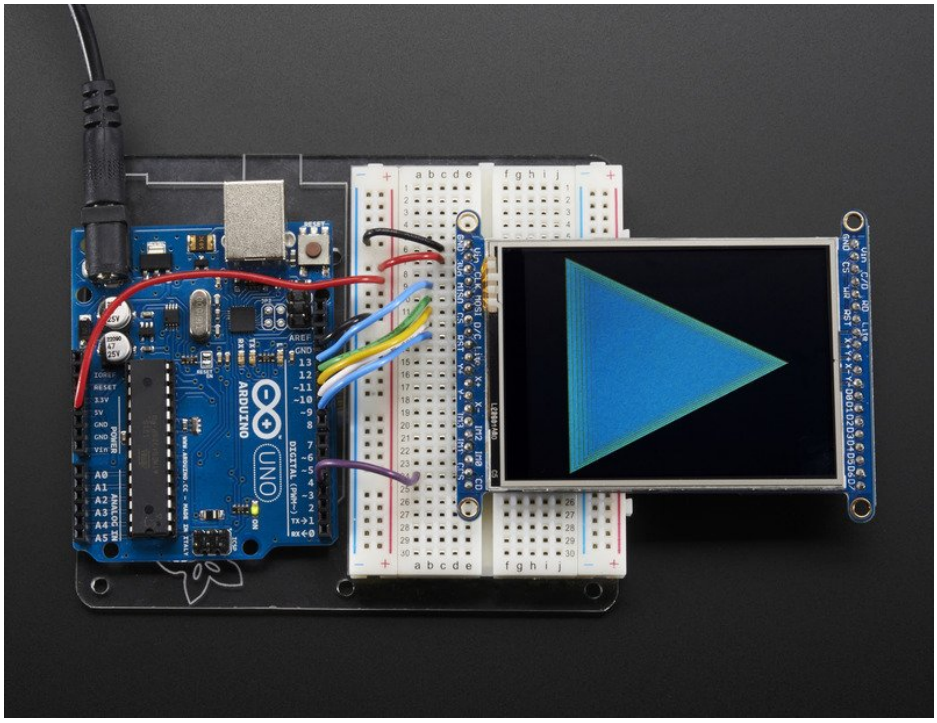
We also have a great tutorial on Arduino library installation at:
http://learn.adafruit.com/adafruit-all-about-arduino-libraries-install-use (https://adafru.it/aYM)



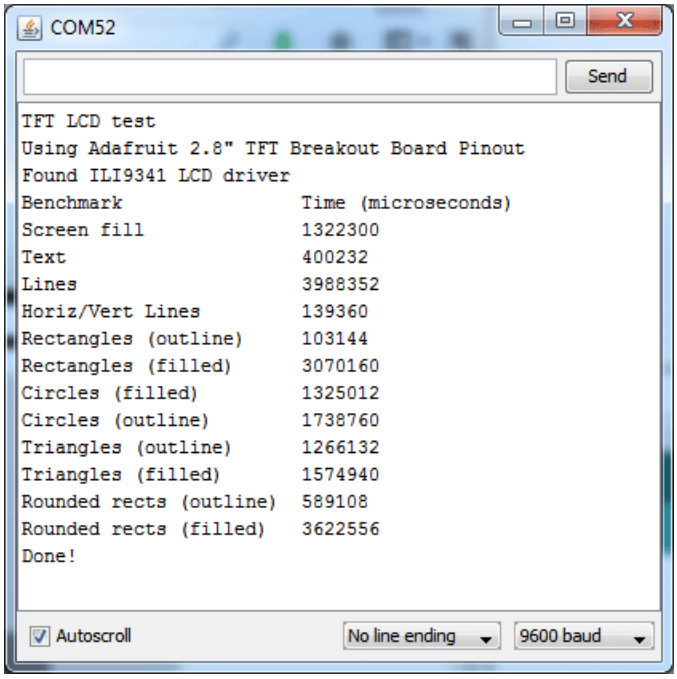
After restarting the Arduino software, you should see a new **example** folder called **Adafruit_TFTLCD** and inside, an example called **graphicstest**. Upload that sketch to your Arduino. You may need to press the Reset button to reset the arduino and TFT. You should see a collection of graphical tests draw out on the TFT.

(The images below shows SPI wiring but the graphical output should be similar!)

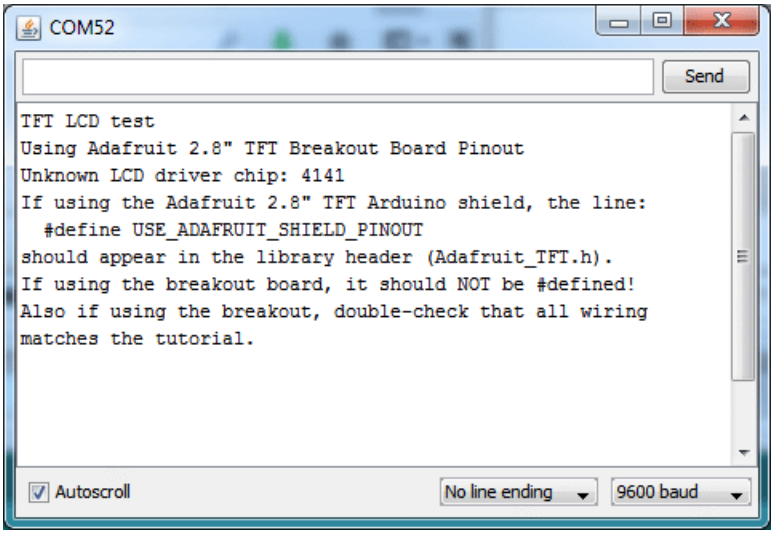




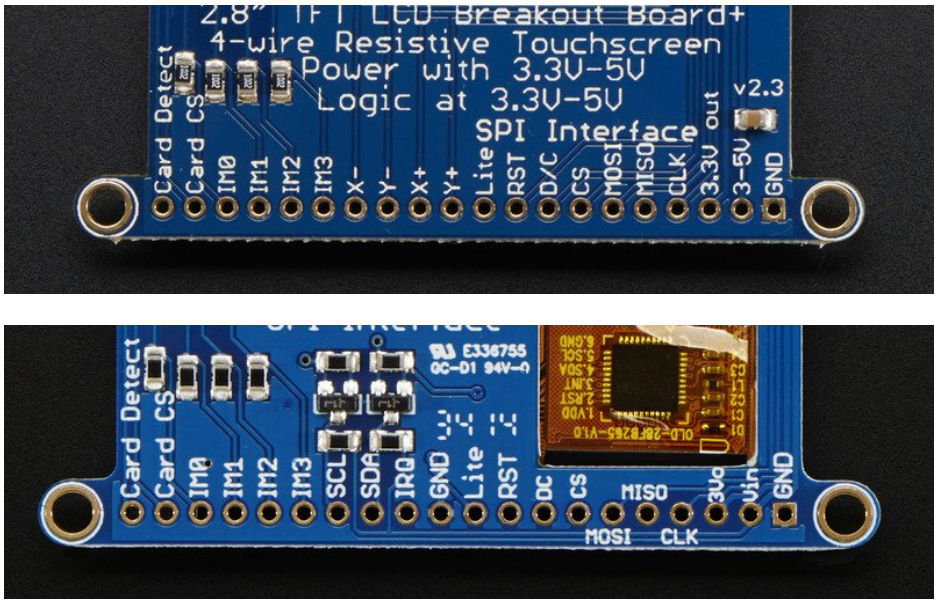
If you're having difficulties, check the serial console.The first thing the sketch does is read the driver code from the TFT. It should be **0x9341** (for the **ILI9341** controller inside)



If you **Unknown Driver Chip** then it's probably something with your wiring, double check and try again!



SPI Wiring and Test

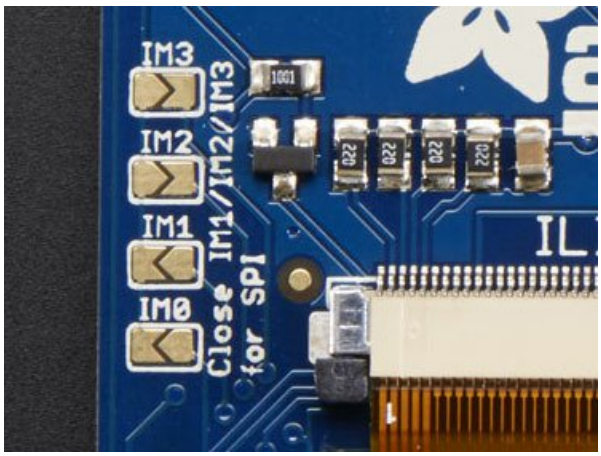


We show the 2.8" version of this breakout in the photos below but the 3.2" TFT is identical, just a lil bit bigger

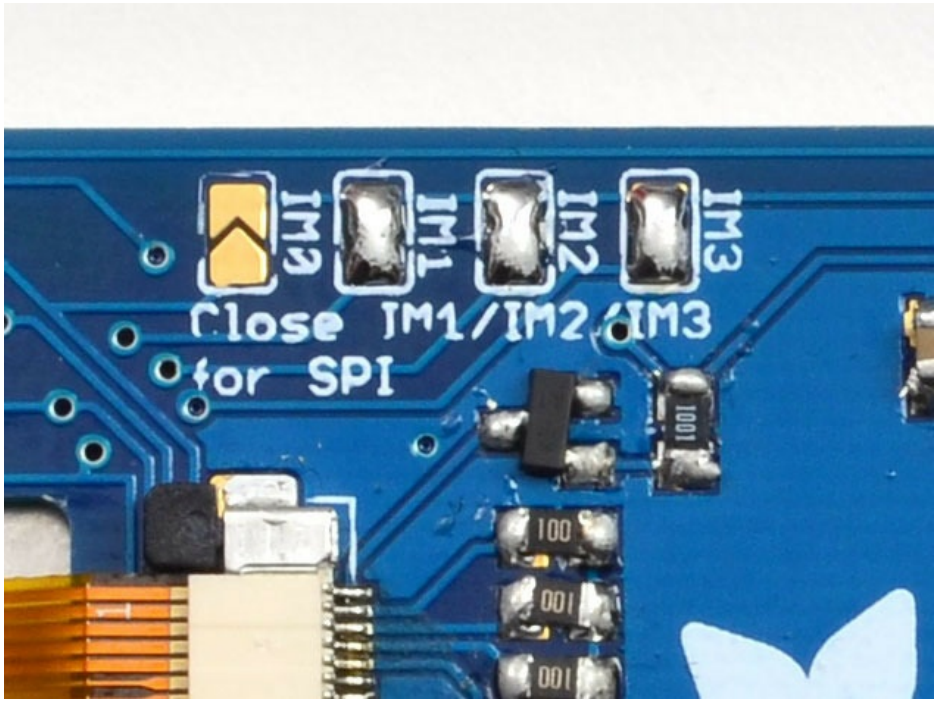
Don't forget, we're using the SPI interface side of the PCB!

SPI Mode Jumpers

Before you start, we'll need to tell the display to put us in SPI mode so it will know which pins to listen to. To do that, we have to connect tbe **IM1**, **IM2** and **IM3** pins to 3.3V. The easiest way to do that is to solder closed the **IMx** jumpers on the back of the PCB. Turn over the PCB and find the solder jumpers



With your soldering iron, melt solder to close the three jumpers indicated **IM1 IM2** and **IM3** (do not solder closed **IM0**!)



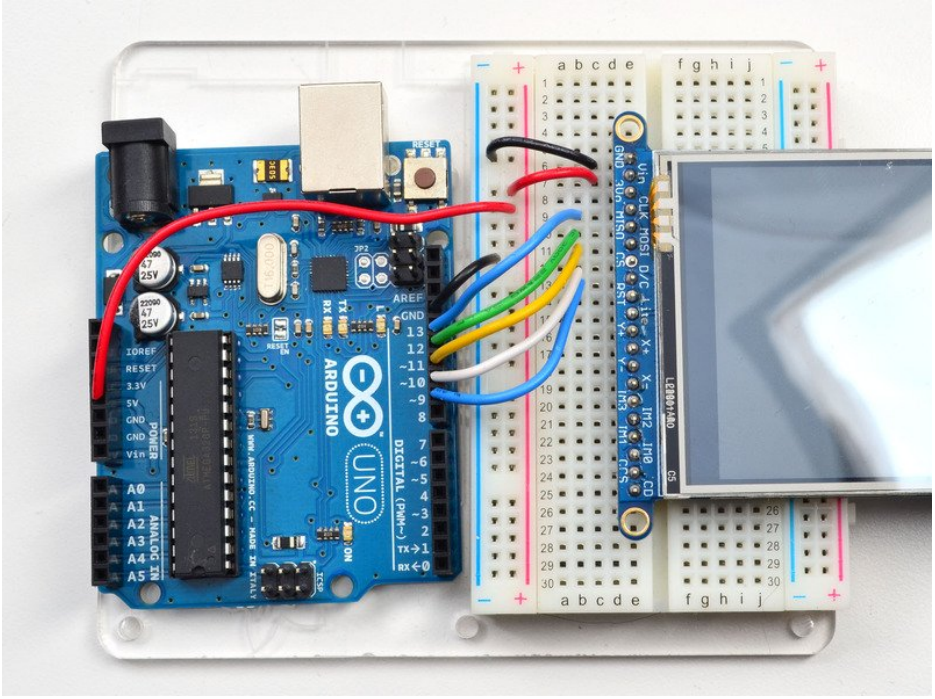
If you really don't want to solder them, you can also wire the breakout pins to the **3vo** pin, just make sure you don't tie them to 5V by accident! For that reason, we suggest going with the solder-jumper route.

Wiring

Wiring up the display in SPI mode is much easier than 8-bit mode since there's way fewer wires. Start by connecting the power pins

- **3-5V Vin** connects to the Arduino **5V** pin
- **GND** connects to Arduino ground
- **CLK** connects to SPI clock. On Arduino Uno/Duemilanove/328-based, thats **Digital 13**. On Mega's, its **Digital 52** and on Leonardo/Due its **ICSP-3** (See SPI Connections for more details (https://adafru.it/d5h))
- **MISO** connects to SPI MISO. On Arduino Uno/Duemilanove/328-based, thats **Digital 12**. On Mega's, its **Digital 50** and on Leonardo/Due its **ICSP-1** (See SPI Connections for more details (https://adafru.it/d5h))
- **MOSI** connects to SPI MOSI. On Arduino Uno/Duemilanove/328-based, thats **Digital 11**. On Mega's, its **Digital 51** and on Leonardo/Due its **ICSP-4** (See SPI Connections for more details (https://adafru.it/d5h))
- **CS** connects to our SPI Chip Select pin. We'll be using **Digital 10** but you can later change this to any pin
- **D/C** connects to our SPI data/command select pin. We'll be using **Digital 9** but you can later change this pin too.

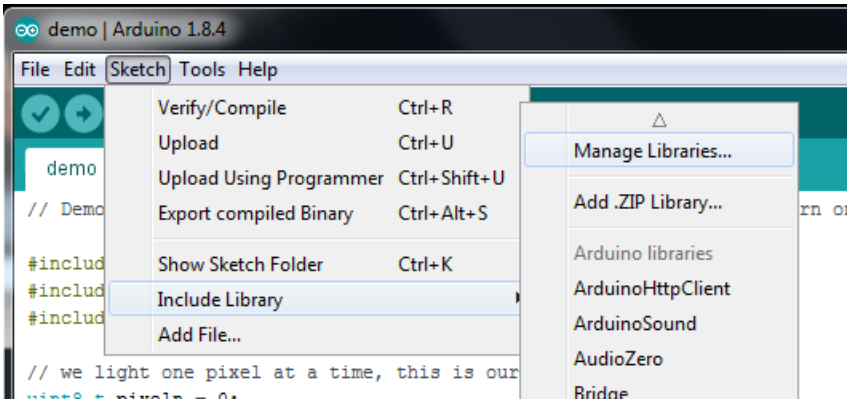
That's it! You do not need to connect the **RST** or other pins for now.



Install Libraries

You'll need a few libraries to use this display

From within the Arduino IDE, open up the **Library Manager...**

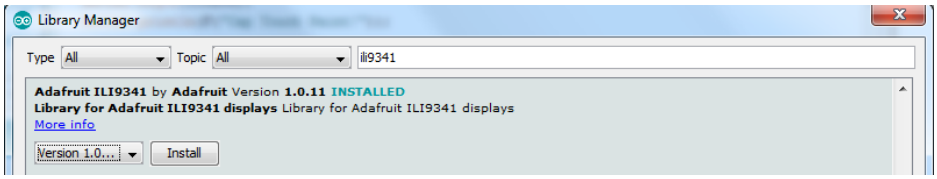


Install Adafruit ILI9341 TFT Library

We have example code ready to go for use with these TFTs.

Two libraries need to be downloaded and installed: first is the Adafruit ILI9341 library (https://adafru.it/d4d) (this contains the low-level code specific to this device), and second is the Adafruit GFX Library (https://adafru.it/aJa) (which handles graphics operations common to many displays we carry). If you have **Adafruit_GFX** already, make sure its the most recent version since we've made updates for better performance

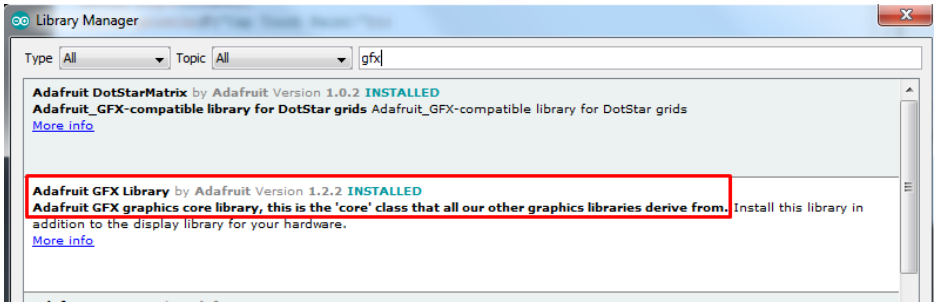
Search for **ILI9341** and install the **Adafruit ILI9341** library that pops up!



For more details, especially for first-time library installers, check out our great tutorial at http://learn.adafruit.com/adafruit-all-about-arduino-libraries-install-use (https://adafru.it/aYM)

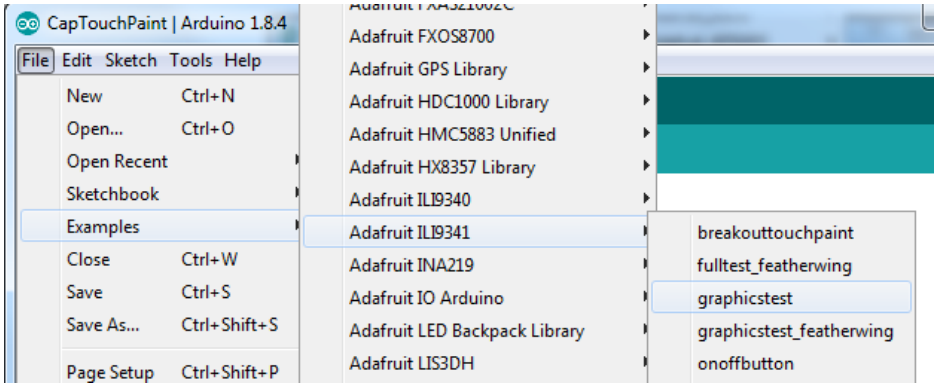
Next up, search for **Adafruit GFX** and locate the core library. A lot of libraries may pop up because we reference it in the description so just make sure you see **Adafruit GFX Library** in bold at the top.

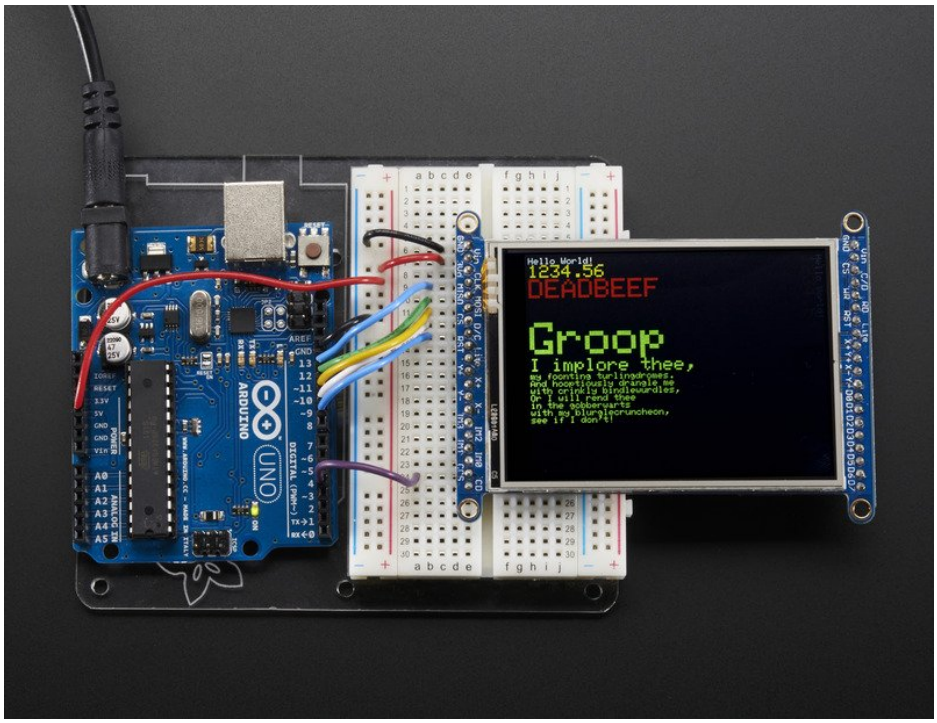
Install it!

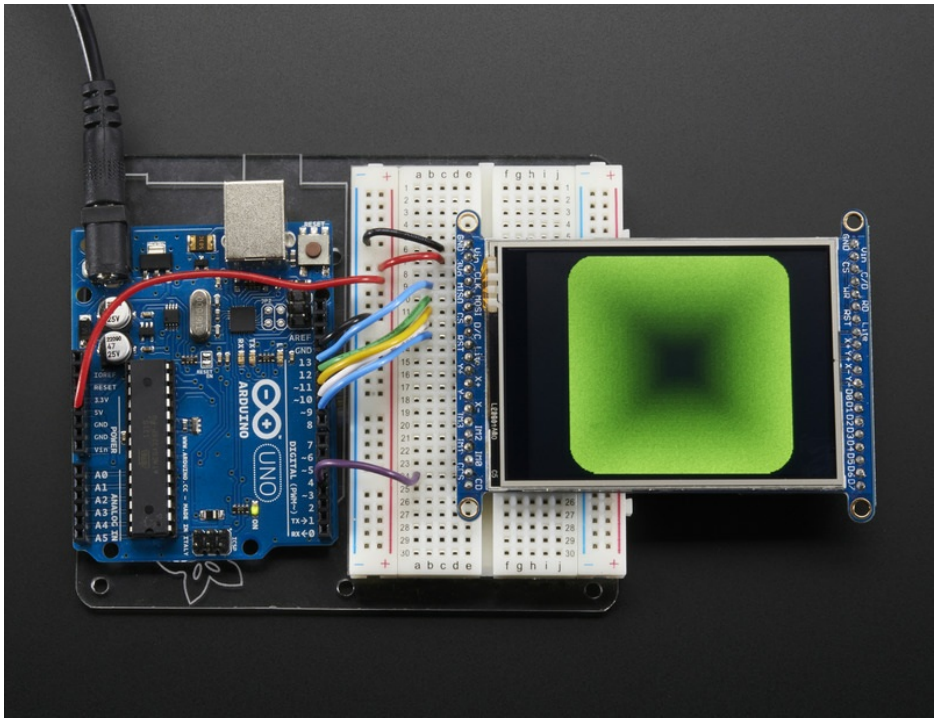


If using an older Arduino IDE (pre-1.8.10), also locate and install **Adafruit_BusIO**.

After restarting the Arduino software, you should see a new **example** folder called **Adafruit_ILI9341** and inside, an example called **graphicstest**. Upload that sketch to your Arduino. You may need to press the Reset button to reset the arduino and TFT. You should see a collection of graphical tests draw out on the TFT.

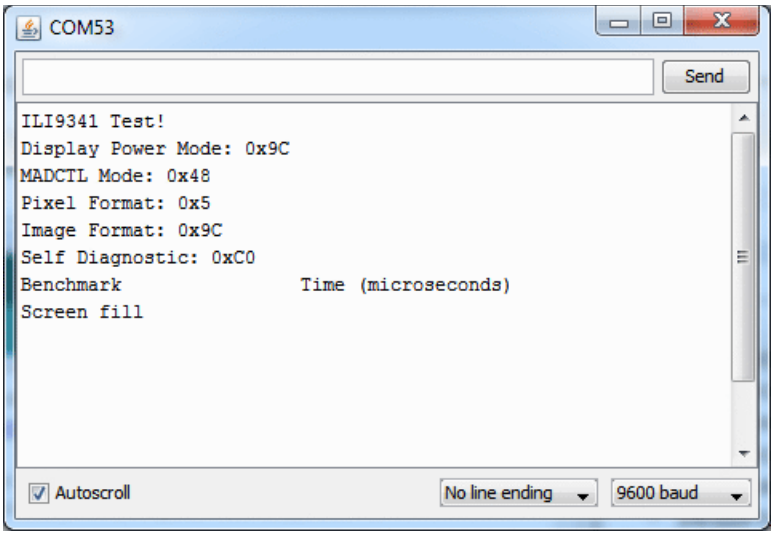






If you're having difficulties, check the serial console.The first thing the sketch does is read the driver configuration from the TFT, you should see the same numbers as below

If you did not connect up the MISO line to the TFT, you wont see the read configuation bytes so please make sure you connect up the MISO line for easy debugging! Once its all working, you can remove the MISO line

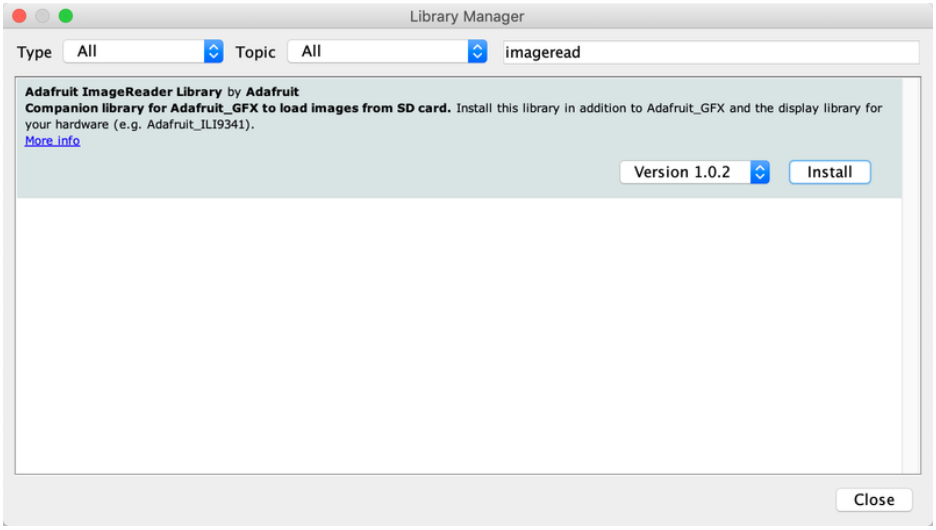

COM53
Send
ILI9341 Test!
Display Power Mode: 0x9C
MADCTL Mode: 0x48
Pixel Format: 0x5
Image Format: 0x9C
Self Diagnostic: 0xC0
Benchmark Time (microseconds)
Screen fill
Autoscroll
No line ending
9600 baud

Bitmaps (SPI Mode)

There is a built in microSD card slot into the breakout, and we can use that to load bitmap images! You will need a microSD card formatted **FAT16 or FAT32** (they almost always are by default).

Its really easy to draw bitmaps. **However, this is only supported when talking to the display in SPI mode, not 8-bit mode!**

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:



Lets start by downloading this image of pretty flowers (pix by johngineer)

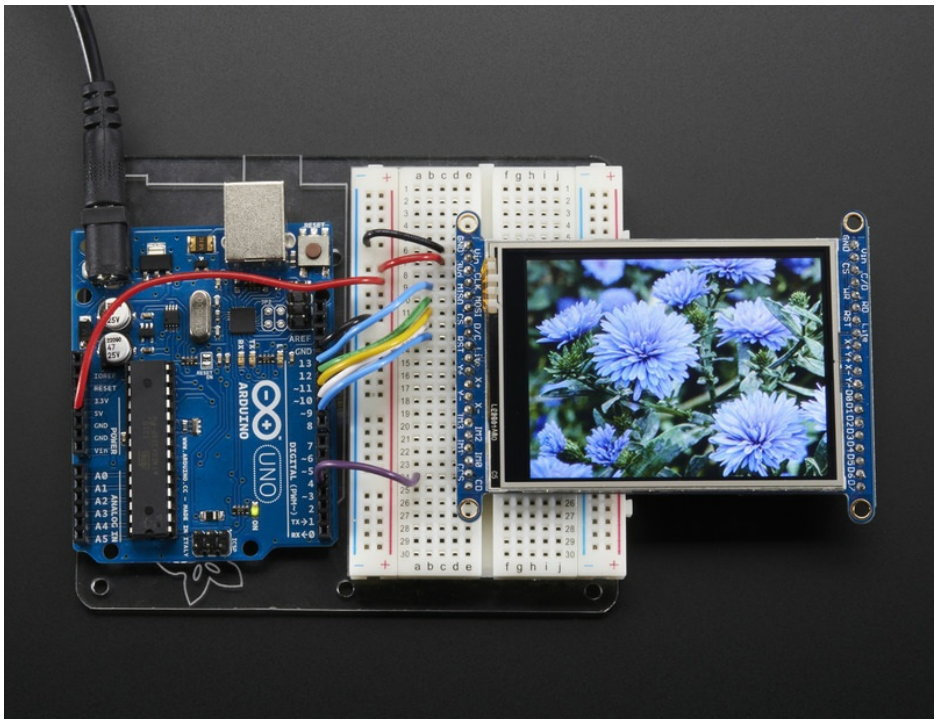
Copy **purple.bmp** into the base directory of a microSD card and insert it into the microSD socket in the breakout.

You'll need to connect up the **CCS** pin to **Digital 4** on your Arduino as well. In the below image, its the extra purple wire

You may want to try the **SD library** examples before continuing, especially one that lists all the files on the SD card

Now upload the **File→examples→Adafruit ImageReader Library→ShieldILI9341** example to your Arduino + breakout. You will see the flowers appear!

We show the 2.8" version of this breakout in the photos below but the 3.2" TFT is identical, just a lil bit bigger



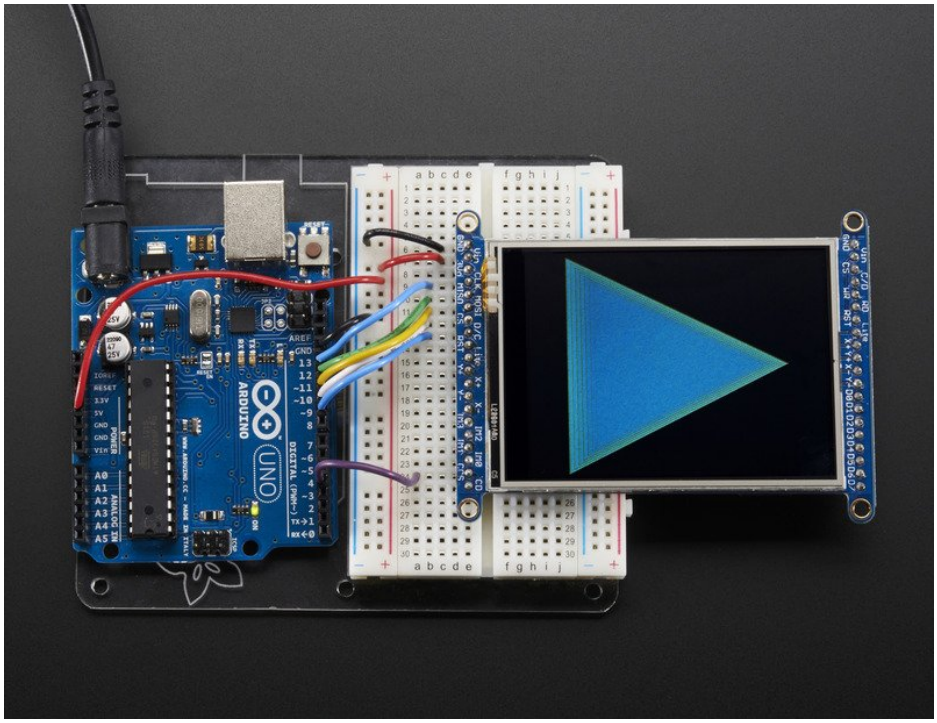
To make new bitmaps, make sure they are less than 240 by 320 pixels and save them in **24-bit BMP format**! They must be in 24-bit format, even if they are not 24-bit color as that is the easiest format for the Arduino. You can rotate images using the **setRotation()** procedure

You can draw as many images as you want - dont forget the names must be less than 8 characters long. Just copy the BMP drawing routines below loop() and call

```
bmpDraw(bmpfilename, x, y);
```

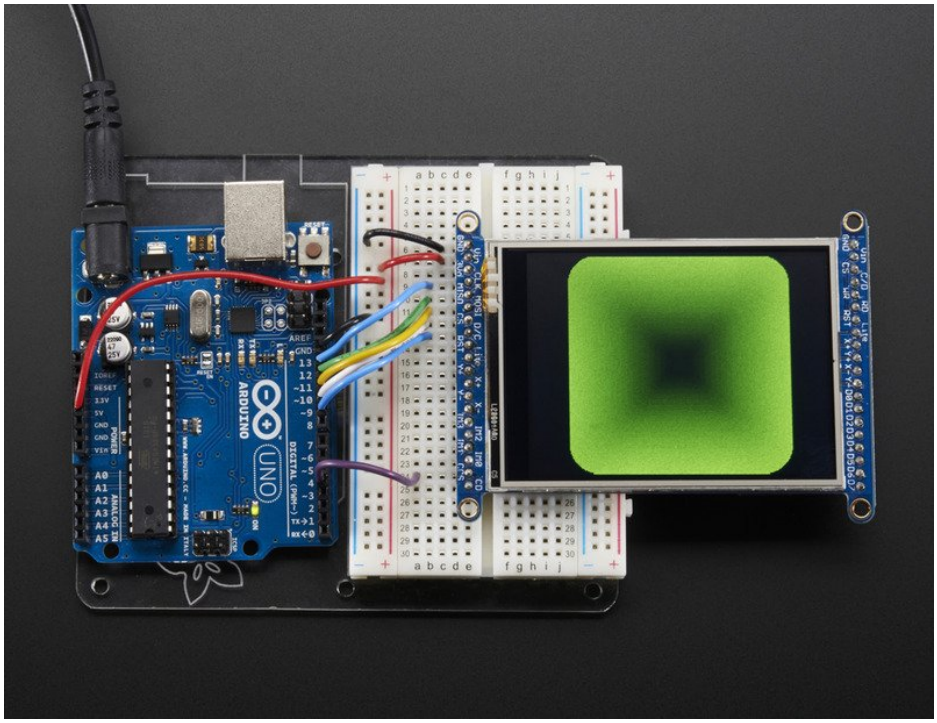
For each bitmap. They can be smaller than 320x240 and placed in any location on the screen.

Adafruit GFX library



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

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



Check out our detailed tutorial here http://learn.adafruit.com/adafruit-gfx-graphics-library (https://adafru.it/aPx)

It covers the latest and greatest of the GFX library. The GFX library is used in both 8-bit and SPI modes so the underlying commands (drawLine() for example) are identical!
(https://adafru.it/aPx)

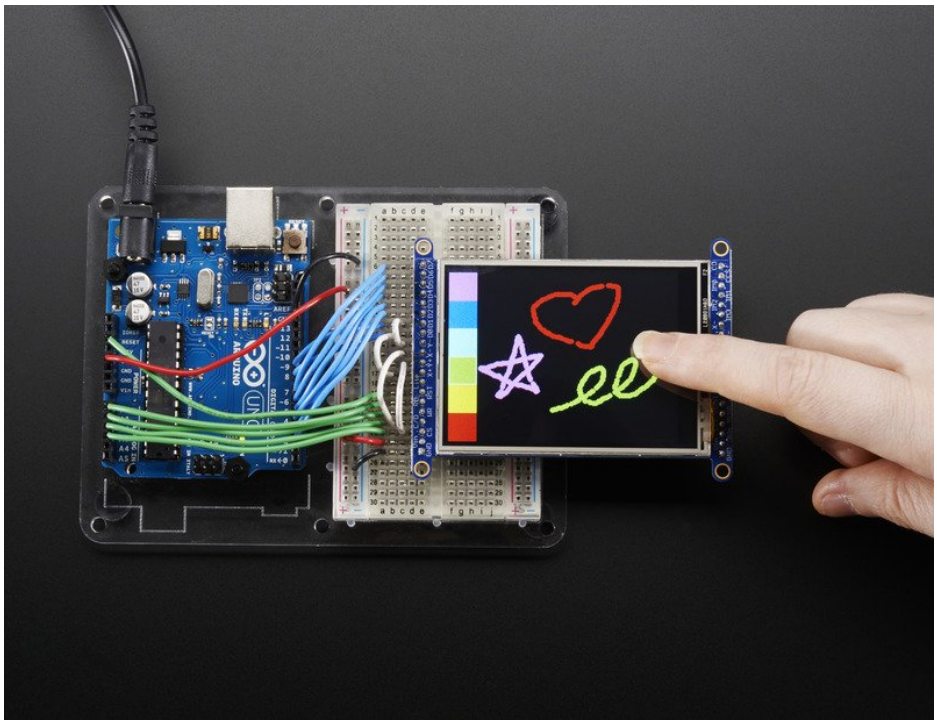
Resistive Touchscreen

The LCD has a 2.8" or 3.2" 4-wire resistive touch screen glued onto it. You can use this for detecting finger-presses, stylus', etc. You'll need 4 pins to talk to the touch panel, and at least 2 must be analog inputs. The touch screen is a completely separate part from the TFT, so be aware if you rotate the display or have the TFT off or reset, the touch screen doesn't "know" about it - its just a couple resistors!

We have a demo for the touchscreen + TFT that lets you 'paint' simple graphics. There's versions for both SPI and 8-bit mode and are included in the libraries. Just make sure you have gone thru the TFT test procedure already since this builds on that.

Remember, if you rotate the screen drawing with setRotation() you'll have to use map() or similar to flip around the X/Y coordinates for the touchscreen as well! It doesn't know about drawing rotation

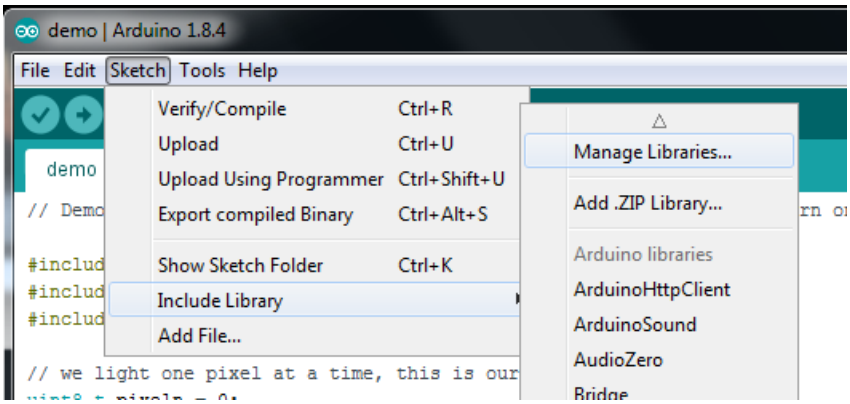
We show the 2.8" version of this breakout in the photos below but the 3.2" TFT is identical, just a lil bit bigger



Download Library

Begin by grabbing our analog/resistive touchscreen library (https://adafru.it/aT1) from the Arduino library manager.

Open up the Arduino library manager:



Search for the **Adafruit TouchScreen** library and install it

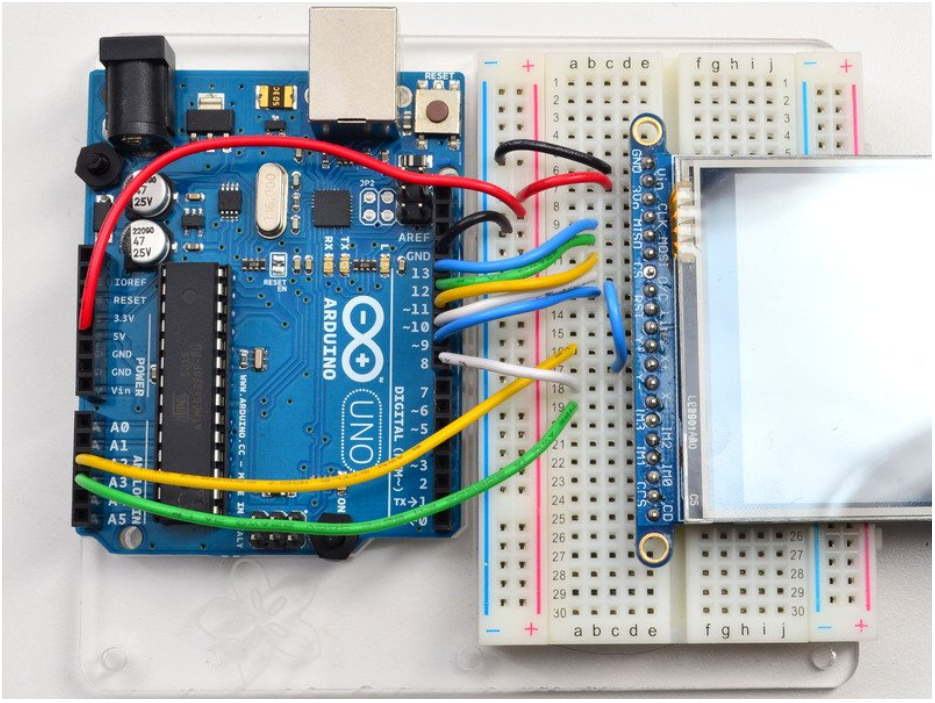
We also have a great tutorial on Arduino library installation at:
http://learn.adafruit.com/adafruit-all-about-arduino-libraries-install-use (https://adafru.it/aYM)

Touchscreen Paint (SPI mode)

An additional 4 pins are required for the touchscreen. For the two analog pins, we'll use **A2** and **A3**. For the other two connections, you can pin any two digital pins but we'll be using **D9** (shared with **D/C**) and **D8** since they are available. We can save the one pin by sharing with **D/C**but you can't share any other SPI pins. So basically you can get away with using only three additional pins.

Wire the additional 4 pins as follows:

- Y+ to Arduino **A2**
- X+ to Arduino **D9** (Same as **D/C**)
- Y- to Arduino **D8**
- X- to Arduino **A3**



Load up the **breakoutTouchPaint** example from the **Adafruit_ILI9341** library and try drawing with your fingernail! You can select colors by touching the 'pallette' of colors on the right

Touchscreen Paint (8-Bit mode)

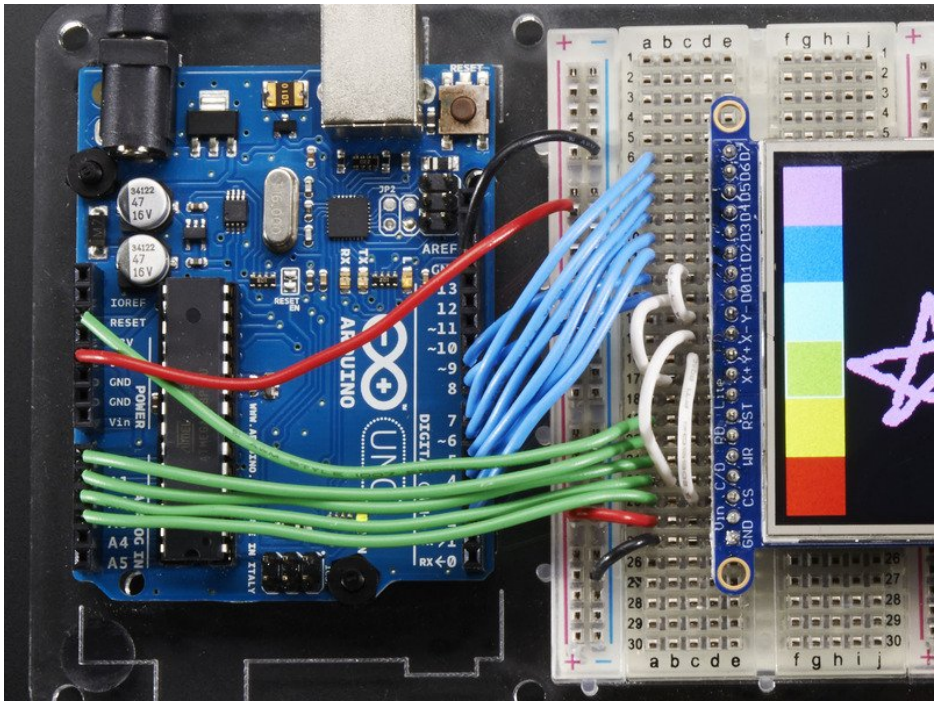
Another 4 pins seems like a lot since already 12 are taken up with the TFT **but** you can **reuse** some of the pins for the TFT LCD! This is because the resistance of the panel is high enough that it doesn't interfere with the digital input/output and we can query the panel in between TFT accesses, when the pins are not being used.

We'll be building on the wiring used in the previous drawing test for UNO

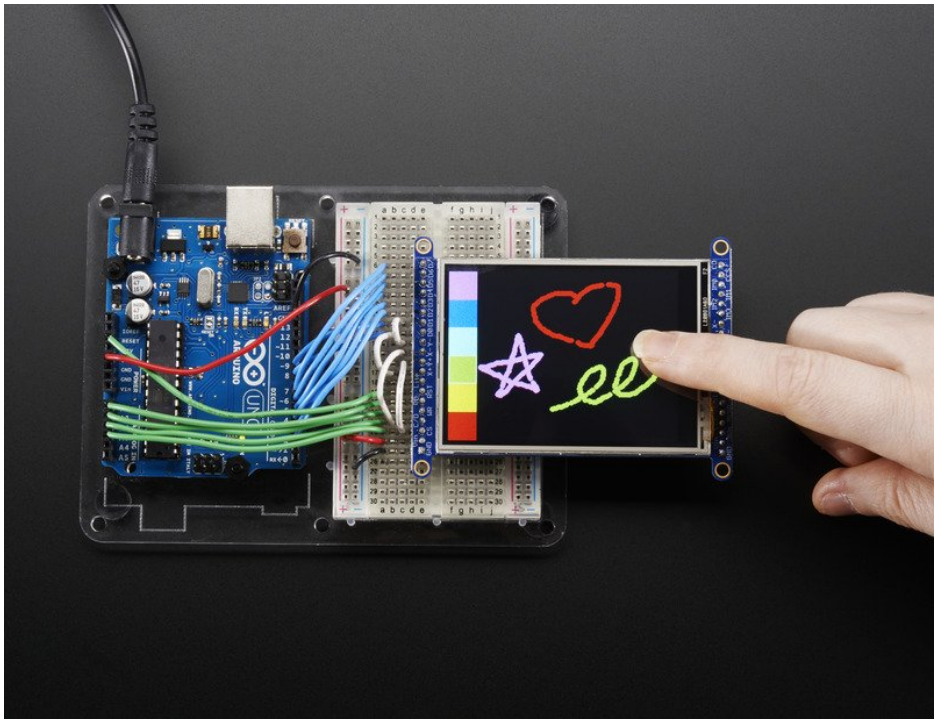
You can wire up the 4 touchscreen pins as follows. Starting from the top

- **Y-** connects to digital **#9** (also **D1**)
- The next one down (**X-**) connects to **Analog 2** (also **C/D**)
- The next one over (**Y+**) connects to **Analog 3** (also **CS**)
- The last one (**X+**) connects to digital **8**. (also **D0**)

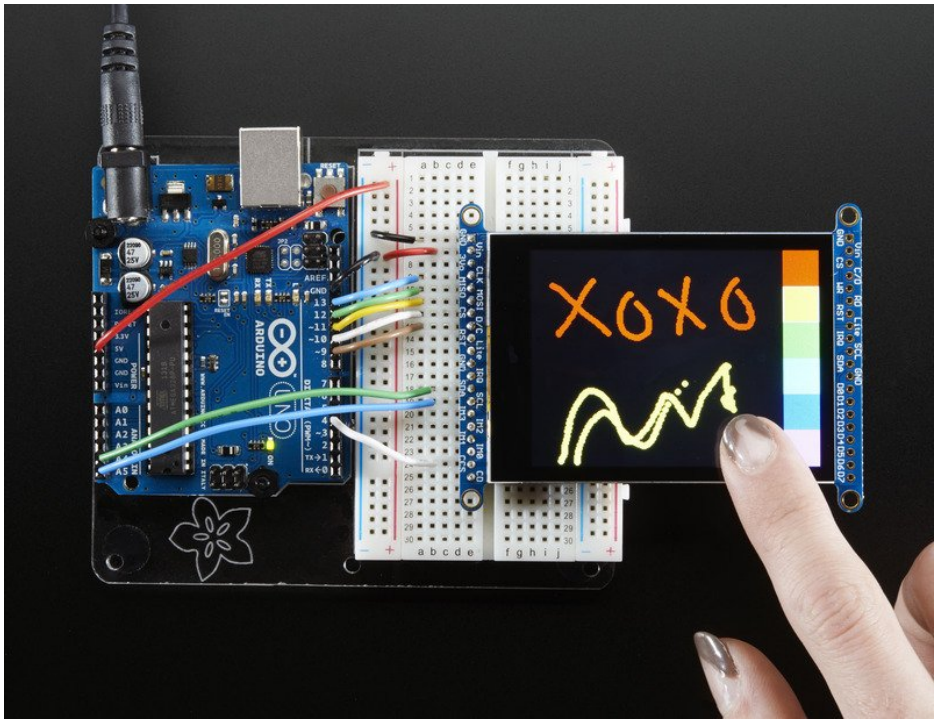
The X- and Y+ pins pretty much have to connect to those analog pins (or to analog 4/5) but Y-/X+ can connect to any digital or analog pins.



Load up the **tftpaint** example from the **Adafruit_TFTLCD** library and try drawing with your fingernail! You can select colors by touching the 'pallette' of colors on the right



Capacitive Touchscreen



We now have a super-fancy capacitive touch screen version of this shield. Instead of a resistive controller that needs calibration and pressing down, the capacitive has a hard glass cover and can be used with a gentle fingertip. It is a single-touch capacitive screen only!

The capacitive touch screen controller communicates over I2C, which uses two hardwire pins. However, you can share these pins with other sensors and displays as long as they don't conflict with I2C address 0x38.

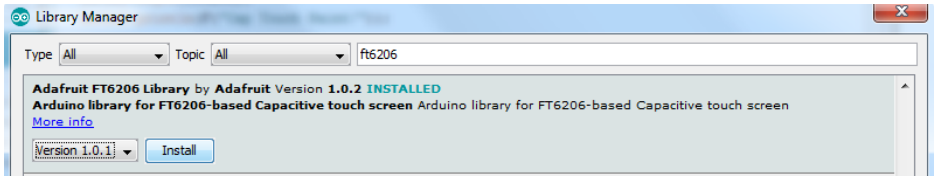
The capacitive touch chip shares the same power and ground as the display, the only new pins you must connect are **SDA** and **SCL** - these must connect to the Arduino I2C pins.

- Connect the **SCL** 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 **SDA** 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**

This demo uses the SPI 'side' of the display so get the SPI drawing demos working before you continue!You can adapt the code for use with the 8-bit side, just instantiate the FT6206 library and see the reference below!

Download the FT6206 Library

To control the touchscreen you'll need one more library (https://adafru.it/dGG) - the FT6206 controller library which does all the low level chatting with the FT6206 driver chip. Use the library manager and search for **FT6206** and select the Adafruit FT6206 library:



Once you have the library installed, restart the IDE. Now from the **examples->Adafruit_FT6206** menu select **CapTouchPaint** and upload it to your Arduino.

The touch screen is made of a thin glass sheet, and its very fragile - a small crack or break will make the entire touch screen unusable. Don't drop or roughly handle the TFT and be especially careful of the corners and edges. When pressing on the touchscreen, remember you cannot use a fingernail, it must be a fingerpad. Do not press harder and harder until the screen cracks!

FT6206 Library Reference

Getting data from the touchscreen is fairly straight forward. Start by creating the touchscreen object with

```
Adafruit_FT6206 ts = Adafruit_FT6206();
```

We're using hardware I2C which is fixed in hardware so no pins are defined.
Then you can start the touchscreen with

```
ts.begin()
```

Check to make sure this returns a True value, which means the driver was found. You can also call **begin(threshvalue)** wish a number from 0-255 to set the touch threshhold. The default works pretty well but if you're having too much sensitivity (or not enought) you can try tweaking it

Now you can call

```
if (ts.touched())
```

to check if the display is being touched, if so call:

```
TS_Point p = ts.getPoint();
```

To get the touch point from the controller. TS_Point has **.x** and **.y** data points. The **x** and **y** points range from 0 to 240 and 0 to 320 respectively. This corresponds to each pixel on the display. The FT6206 does not need to be 'calibrated' but it also doesn't know about rotation. **So if you want to rotate the screen you'll need to manually rotate the x/y points!**

Touchscreen Interrupt pin

Advanced users may want to get an interrupt on a pin (or even, just test a pin rather than do a full SPI query) when the touchscreen is pressed. That's the IRQ pin, which is a 3V logic output from the breakout, you can connect it to any interrupt pin and use it like a 'button press' interrupt. We find that querying/polling the chip is fast enough for most

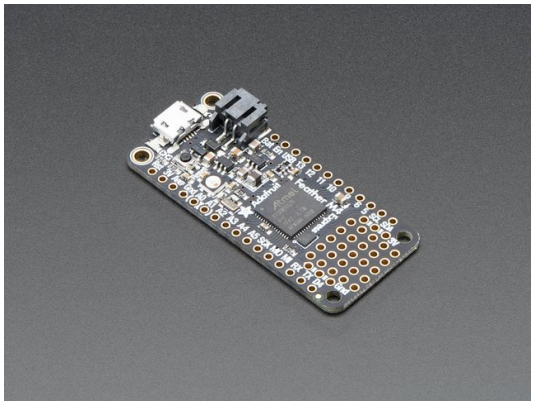
beginner Arduino projects!

FT6206 Library Reference

FT6206 Library Reference (https://adafru.it/Atz)

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 (https://adafru.it/EEm) guide.



Adafruit Feather M4 Express - Featuring ATSAMD51

OUT OF STOCK

Out Of Stock

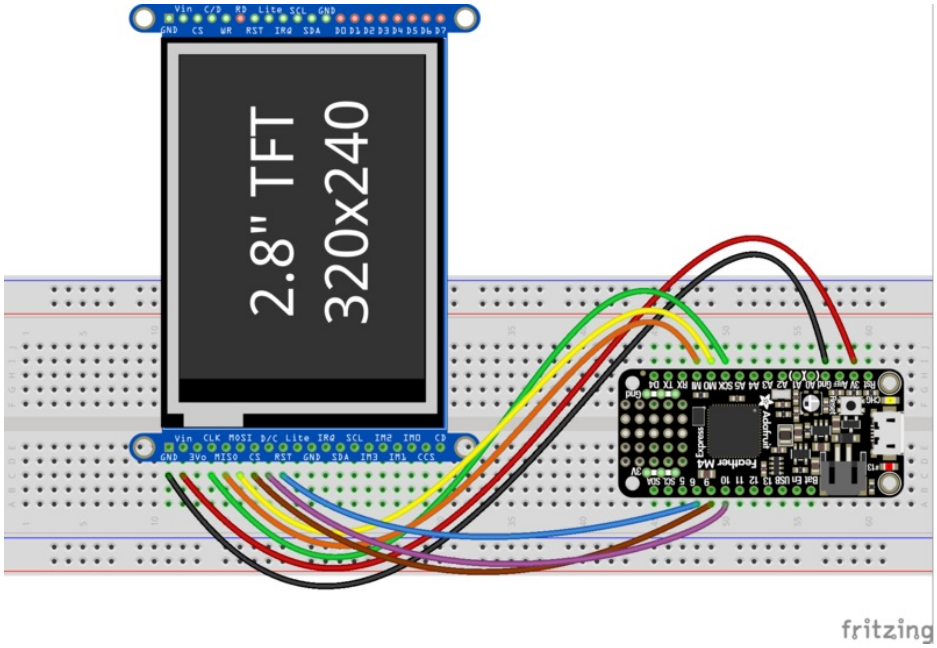
For this guide, we'll assume you have a Feather M4 Express. The steps should be about the same for the Feather M0 Express. To start, if you haven't already done so, follow the assembly instructions for the Feather M4 Express in our Feather M4 Express guide (https://adafru.it/EEm).

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 (https://adafru.it/drl). Also, follow the SPI Wiring and Test page of this guide to be sure your display is setup for SPI. After that, the breakout should be ready to go.

Wiring the Breakout to the Feather

- **3-5V Vin** connects to the Feather **3V** pin
- **GND** connects toFeather ground
- **CLK** connects to SPI clock. On the Feather that's **SCK**.
- **MISO** connects to SPI MISO. On the Feather that's **MI**
- **MOSI** connects to SPI MOSI. On the Feather that's **MO**
- **CS** connects to our SPI Chip Select pin. We'll be using **Digital 9** but you can later change this to any pin
- **D/C** connects to our SPI data/command select pin. We'll be using **Digital 10** but you can later change this pin too.
- **RST** connects to our reset pin. We'll be using **Digital 6** but you can later change this pin too.



https://adafru.it/Fyk

https://adafru.it/Fyk

Required CircuitPython Libraries

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

https://adafru.it/EGe

https://adafru.it/EGe

First, make sure you are running the latest version of Adafruit CircuitPython (https://adafru.it/Amd) 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 (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.

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

- **adafruit_ili9341**

Before continuing make sure your board's lib folder or root filesystem has the **adafruit_ili9341** 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.

https://adafru.it/FiA

https://adafru.it/FiA

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

```
"""
This test will initialize the display using displayio and draw a solid green
background, a smaller purple rectangle, and some yellow text. All drawing is done
using native displayio modules.

Pinouts are for the 2.4" TFT FeatherWing or Breakout with a Feather M4 or M0.
"""
import board
import terminalio
import displayio
from adafruit_display_text import label
import adafruit_ili9341

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

spi = board.SPI()
tft_cs = board.D9
tft_dc = board.D10

display_bus = displayio.FourWire(
    spi, command=tft_dc, chip_select=tft_cs, reset=board.D6
)
display = adafruit_ili9341.ILI9341(display_bus, width=320, height=240)

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

# Draw a green background
color_bitmap = displayio.Bitmap(320, 240, 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(280, 200, 1)
inner_palette = displayio.Palette(1)
inner_palette[0] = 0xAA0088  # Purple
inner_sprite = displayio.TileGrid(inner_bitmap, pixel_shader=inner_palette, x=20, y=20)
splash.append(inner_sprite)

# Draw a label
text_group = displayio.Group(max_size=10, scale=3, x=57, y=120)
text = "Hello World!"
text_area = label.Label(terminalio.FONT, text=text, color=0xFFFF00)
text_group.append(text_area)  # Subgroup for text scaling
splash.append(text_group)

while True:
    pass
```

Code Details

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_ili9341` driver.

```
import board
import displayio
import terminalio
from adafruit_display_text import label
import adafruit_ili9341
```

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. Next we set the Chip Select and Data/Command pins that will be used.

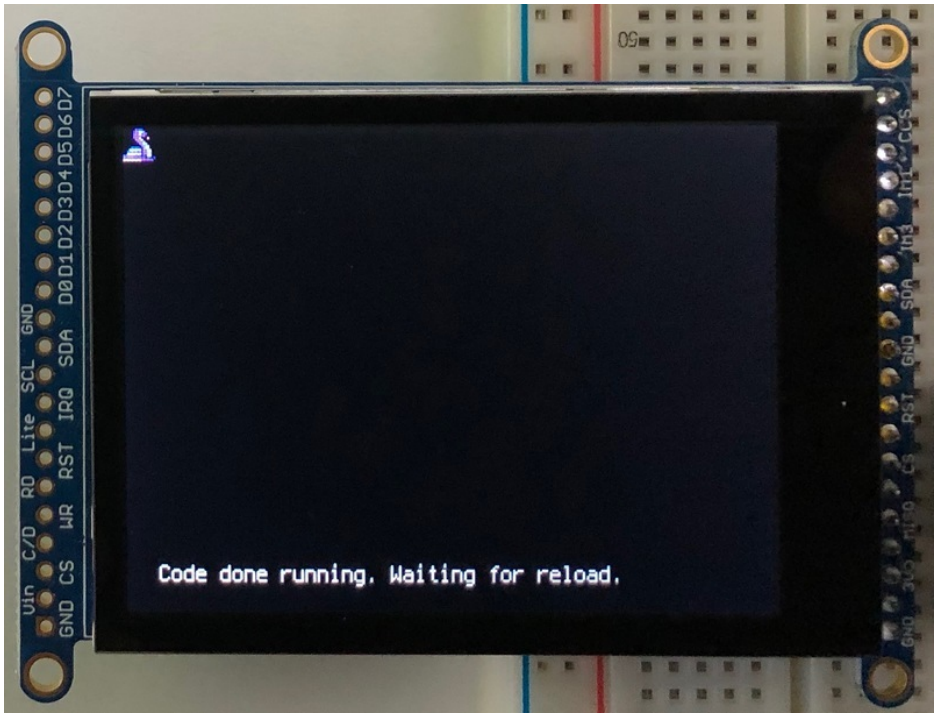
```
spi = board.SPI()
tft_cs = board.D9
tft_dc = board.D10
```

In the next line, we set the display bus to FourWire which makes use of the SPI bus.

```
display_bus = displayio.FourWire(spi, command=tft_dc, chip_select=tft_cs, reset=board.D6)
```

Finally, we initialize the driver with a width of 320 and a height of 240. If we stopped at this point and ran the code, we would have a terminal that we could type at and have the screen update.

```
display = adafruit_ili9341.ILI9341(display_bus, width=320, height=240)
```



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(320, 240, 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)
```

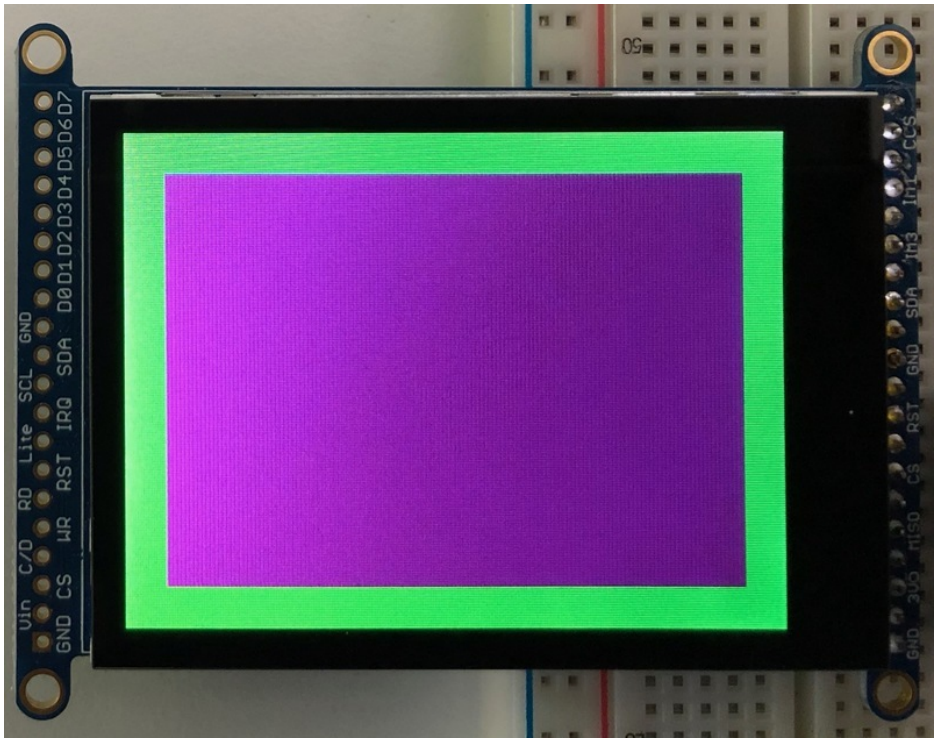
This creates a solid green background which we will draw on top of.

Next we will create a smaller purple rectangle. The easiest way to do this is the 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 20 pixels smaller on each side. The screen is 320x240, so we'll want to subtract 40 from each of those numbers.

We'll also want to place it at the position `(20, 20)` so that it ends up centered.

```
inner_bitmap = displayio.Bitmap(280, 200, 1)
inner_palette = displayio.Palette(1)
inner_palette[0] = 0xAA0088 # Purple
inner_sprite = displayio.TileGrid(inner_bitmap,
                                  pixel_shader=inner_palette,
                                  x=20, y=20)
splash.append(inner_sprite)
```

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



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

Labels are centered vertically, so we'll place it at 120 for the Y coordinate, and around 57 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_group = displayio.Group(max_size=10, scale=3, x=57, y=120)
text = "Hello World!"
text_area = label.Label(terminalio.FONT, text=text, color=0xFFFF00)
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
```

Using Touch

We won't be covering how to use the touchscreen with CircuitPython in this guide, but the libraries required to use it are:

- For enabling capacitive touch use the Adafruit_CircuitPython_FocalTouch (https://adafru.it/Fsy) library.
- For enabling resistive touch use the Adafruit_CircuitPython_STMPE610 (https://adafru.it/Fsz) library.

Where to go from here

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

Python Wiring and Setup

Wiring

It's easy to use display breakouts with Python and the Adafruit CircuitPython RGB Display (https://adafru.it/u1C) 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 (https://adafru.it/BSN).

Connect the display as shown below to your Raspberry Pi.

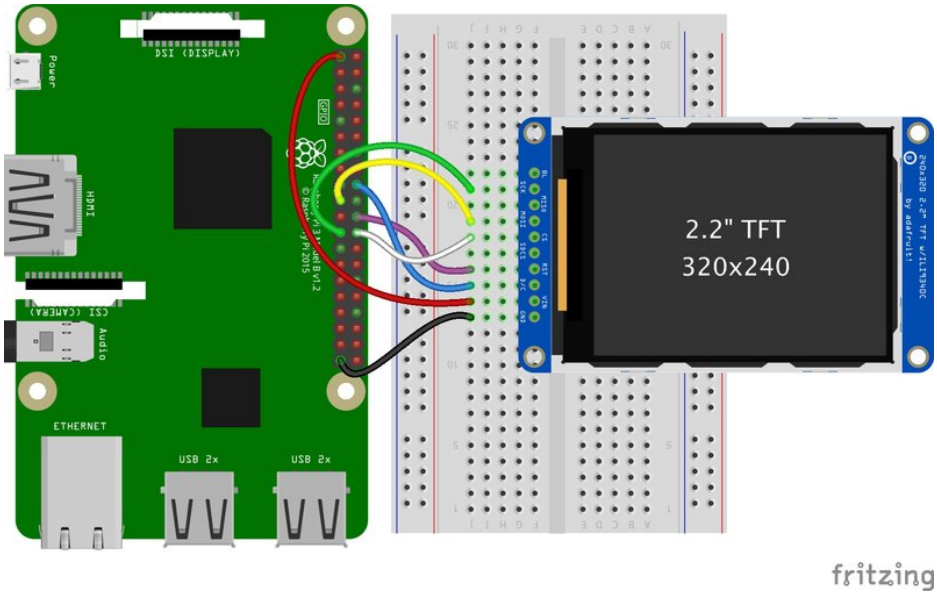
Note this is not a kernel driver that will let you have the console appear on the TFT. However, this is handy when you can't install an fbtft driver, and want to use the TFT purely from 'user Python' code!

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, thats **SLCK**
- **MOSI** connects to SPI MOSI. On the Raspberry Pi, thats 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**



https://adafru.it/H6C

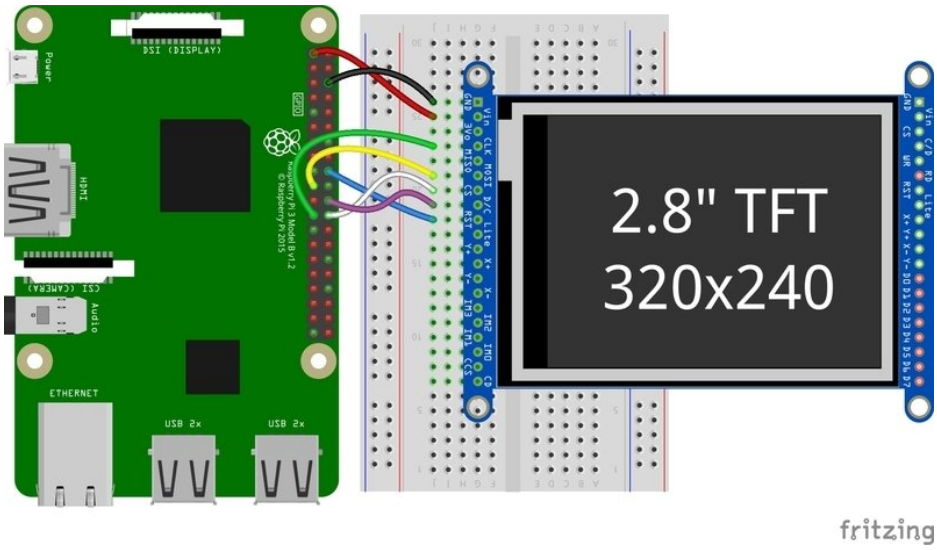
https://adafru.it/H6C

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, thats **SLCK**
- **MOSI** connects to SPI MOSI. On the Raspberry Pi, thats 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.



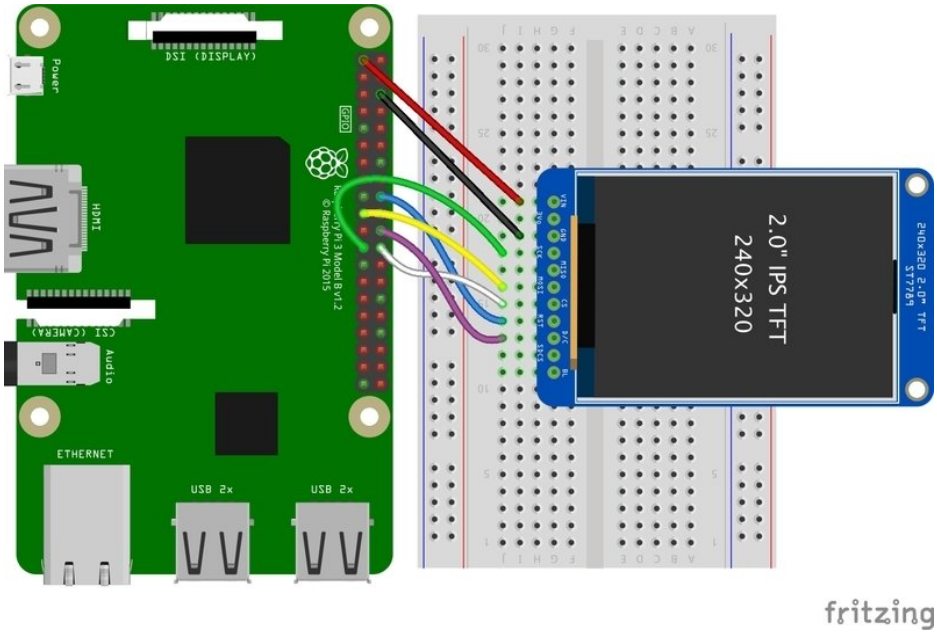
https://adafru.it/H7a

https://adafru.it/H7a

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, thats **SLCK**
- **MOSI** connects to SPI MOSI. On the Raspberry Pi, thats 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.

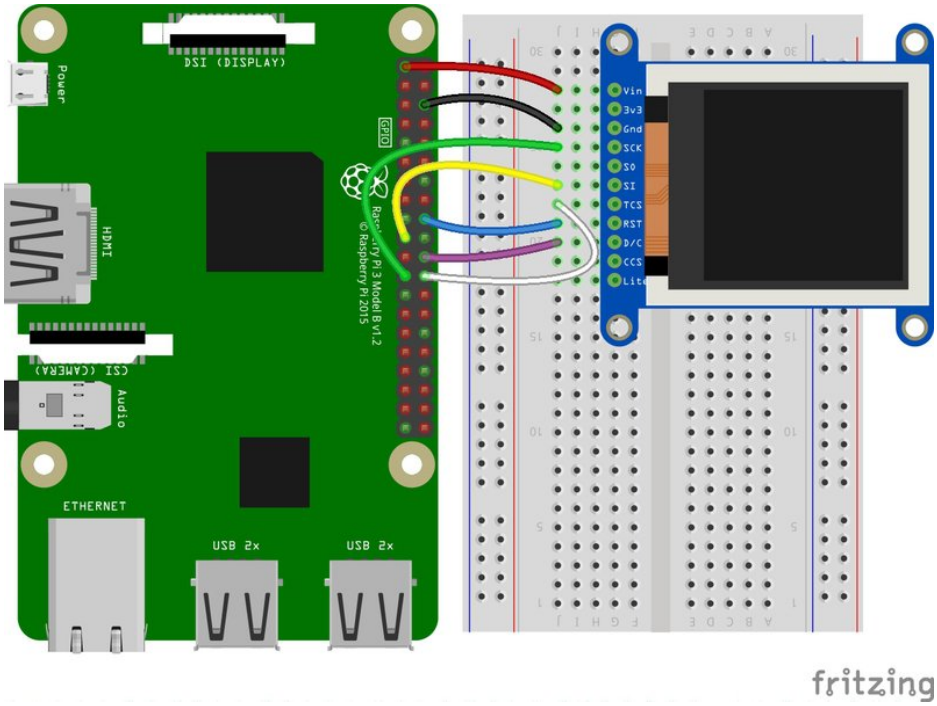


https://adafru.it/H7d

https://adafru.it/H7d

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, thats **SLCK**
- **MOSI** connects to SPI MOSI. On the Raspberry Pi, thats 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.

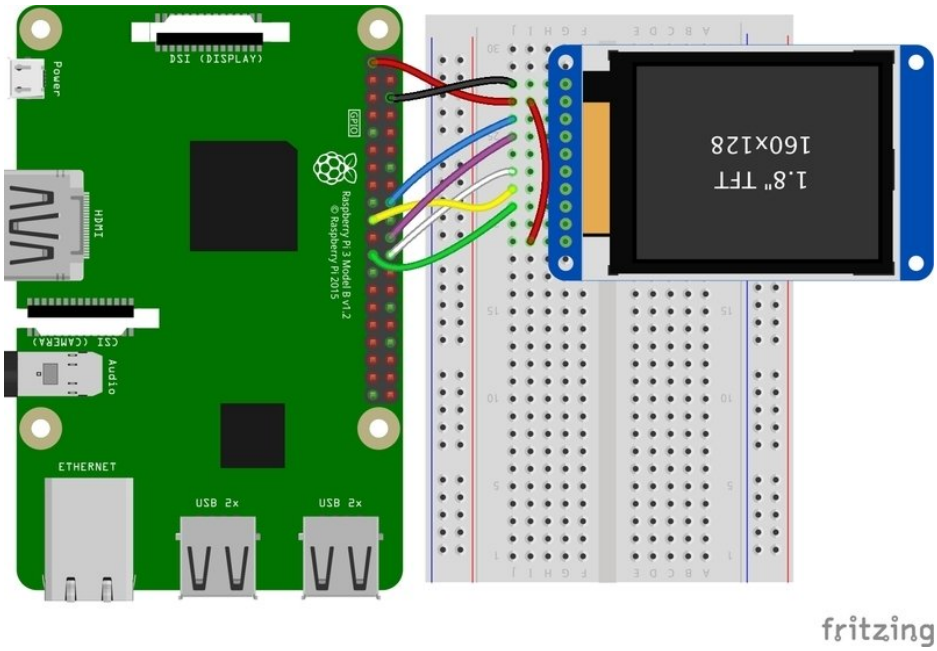


https://adafru.it/H7B

https://adafru.it/H7B

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, thats also **MOSI**
- **CLK** connects to SPI clock. On the Raspberry Pi, thats **SLCK**
- **LITE** connects to the Raspberry Pi's **3V** pin. This can be used to separately control the backlight.



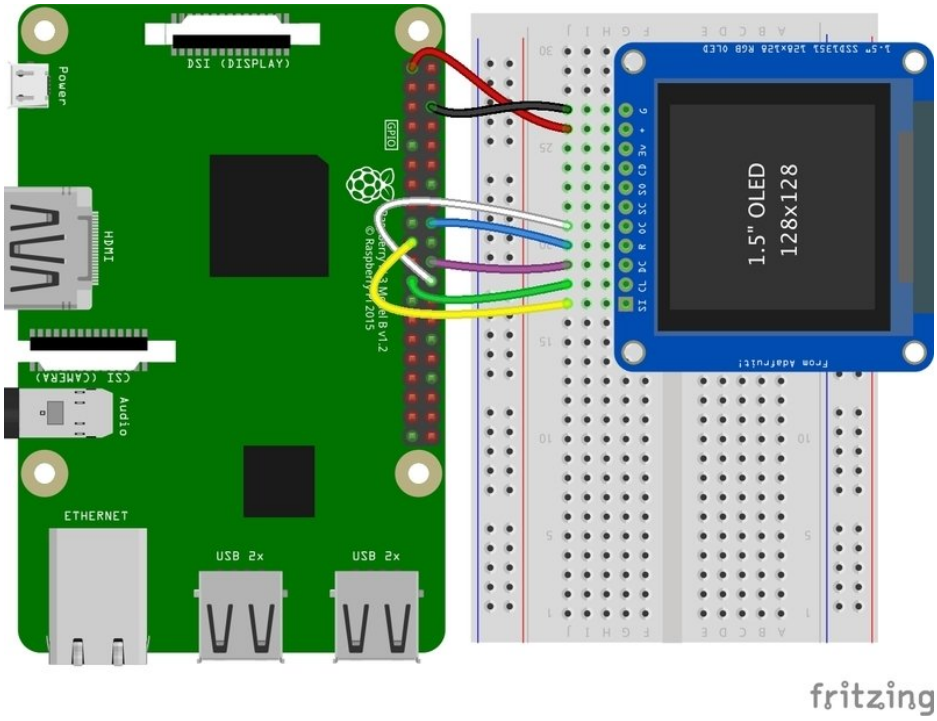
https://adafru.it/H8a

https://adafru.it/H8a

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, thats **SLCK**
- **MOSI** connects to SPI MOSI. On the Raspberry Pi, thats 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.



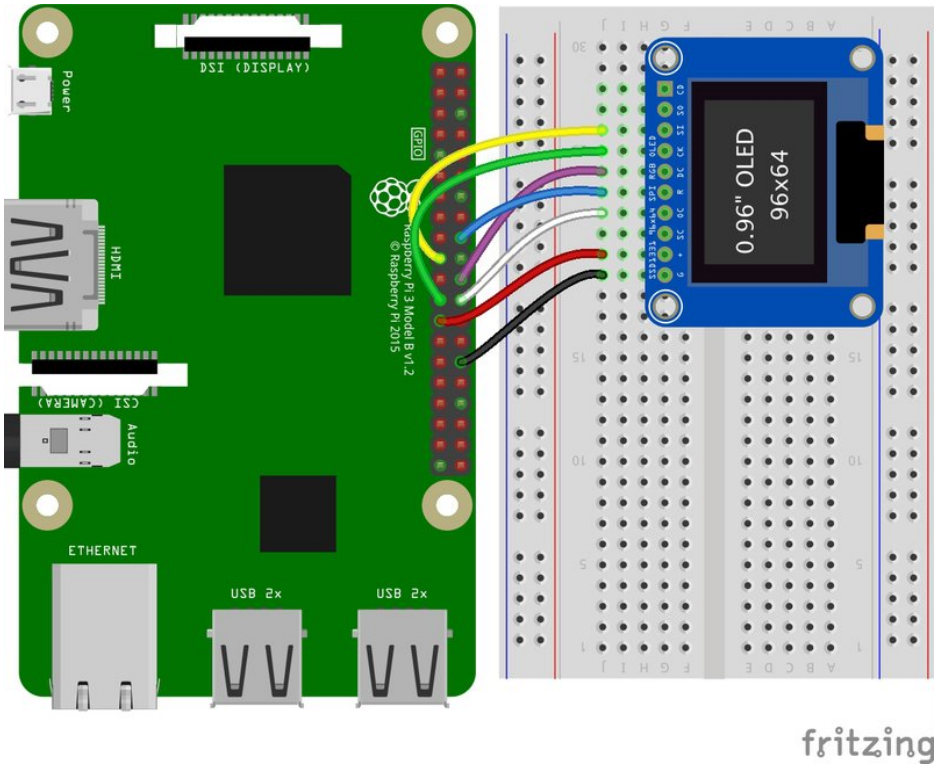
https://adafru.it/H8e

https://adafru.it/H8e

SSD1331-based Display

0.96" OLED Display

- **MOSI** connects to SPI MOSI. On the Raspberry Pi, thats also **MOSI**
- **CLK** connects to SPI clock. On the Raspberry Pi, thats **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**



https://adafru.it/H8D

https://adafru.it/H8D

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 (https://adafru.it/BSN)!

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 ttf-dejavu`

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`

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.

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 Blinka. 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.

```
import digitalio
import board
from PIL import Image, ImageDraw
import adafruit_rgb_display.ili9341 as ili9341
import adafruit_rgb_display.st7789 as st7789  # pylint: disable=unused-import
import adafruit_rgb_display.hx8357 as hx8357  # pylint: disable=unused-import
import adafruit_rgb_display.st7735 as st7735  # pylint: disable=unused-import
import adafruit_rgb_display.ssd1351 as ssd1351  # pylint: disable=unused-import
import adafruit_rgb_display.ssd1331 as 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 = 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,                 # 0.96" MiniTFT ST7735R
# 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.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))
```

```
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 out 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 the line that starts with:

`disp = ili9341.ILI9341(spi,`

and uncomment the line appropriate for your display. The displays have a rotation property so that it can be set in just one place.

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

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

#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,                # 0.96" MiniTFT ST7735R
#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 at 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.

```
import digitalio
import board
from PIL import Image, ImageDraw, ImageFont
import adafruit_rgb_display.ili9341 as ili9341
import adafruit_rgb_display.st7789 as st7789  # pylint: disable=unused-import
import adafruit_rgb_display.hx8357 as hx8357  # pylint: disable=unused-import
import adafruit_rgb_display.st7735 as st7735  # pylint: disable=unused-import
import adafruit_rgb_display.ssd1351 as ssd1351  # pylint: disable=unused-import
import adafruit_rgb_display.ssd1331 as 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 = 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,                 # 0.96" MiniTFT ST7735R
# 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:

```
import time
import subprocess
import digitalio
import board
from PIL import Image, ImageDraw, ImageFont
import adafruit_rgb_display.ili9341 as ili9341
import adafruit_rgb_display.st7789 as st7789  # pylint: disable=unused-import
import adafruit_rgb_display.hx8357 as hx8357  # pylint: disable=unused-import
```

```
import adafruit_rgb_display.st7735 as st7735  # pylint: disable=unused-import
import adafruit_rgb_display.ssd1351 as ssd1351  # pylint: disable=unused-import
import adafruit_rgb_display.ssd1331 as 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 = 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,                 # 0.96" MiniTFT ST7735R
# 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`

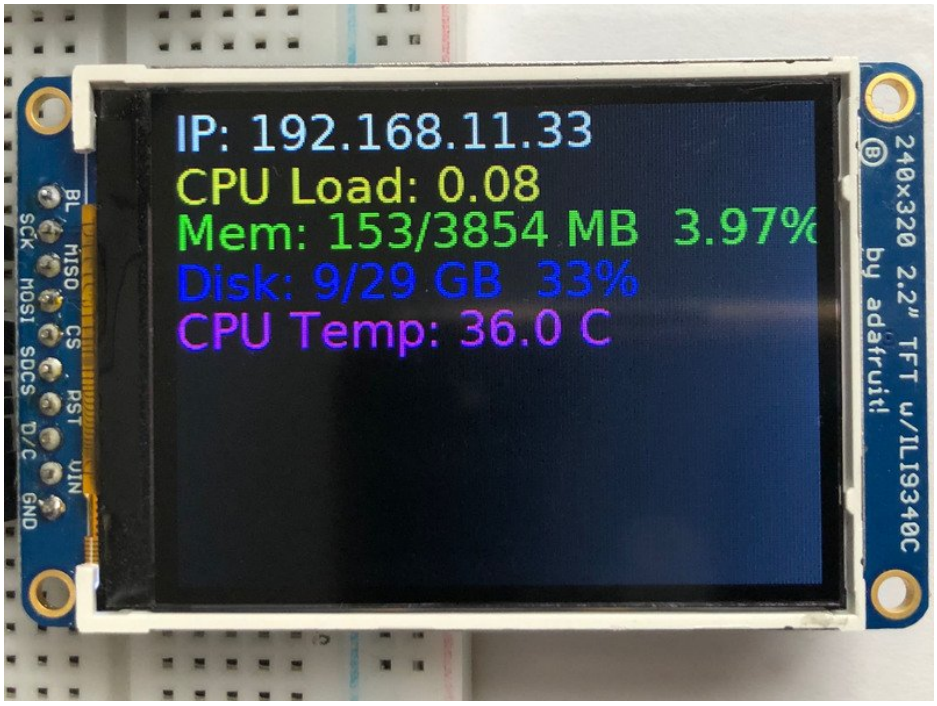
```
cmd = "hostname -I | cut -d\' \' -f1"
IP = "IP: "+subprocess.check_output(cmd, shell=True).decode("utf-8")
cmd = "top -bn1 | grep load | awk '{printf \"CPU Load: %.2f\", $(NF-2)}'"
CPU = subprocess.check_output(cmd, shell=True).decode("utf-8")
cmd = "free -m | awk 'NR==2{printf \"Mem: %s/%s MB  %.2f%%\", $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)
```

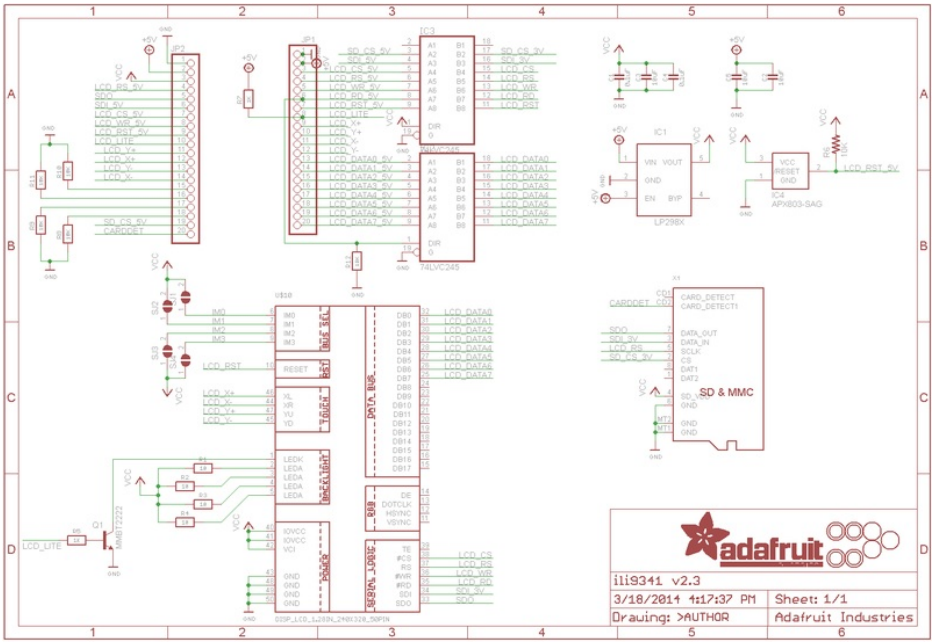


Downloads

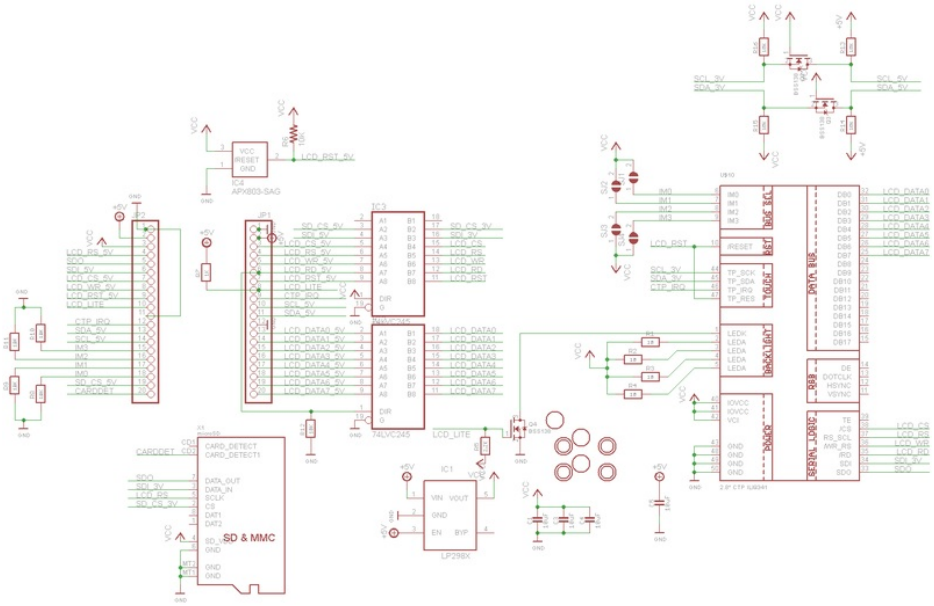
Datasheets & Files

- ILI9341 TFT controller chip datasheet (https://adafru.it/d4l) (this is what you want to refer to if porting or if you want to look at the TFT command set)
- Raw 2.8" Resistive TFT datasheet (https://adafru.it/sEt)
- Raw 3.2" Resistive TFT datasheet (https://adafru.it/Edc)
- Raw 2.8" Capacitive TFT datasheet (https://adafru.it/rwA)
- FT6206 Datasheet (https://adafru.it/sEu) & App note (https://adafru.it/dRn) (capacitive chip)
- Fritzing objects in Adafruit Fritzing Library (https://adafru.it/c7M)
- 2.8" TFT with Capacitive Touch EagleCAD files (https://adafru.it/pAr)
- 2.8" TFT with Resistive Touch EagleCAD files (https://adafru.it/pAs)

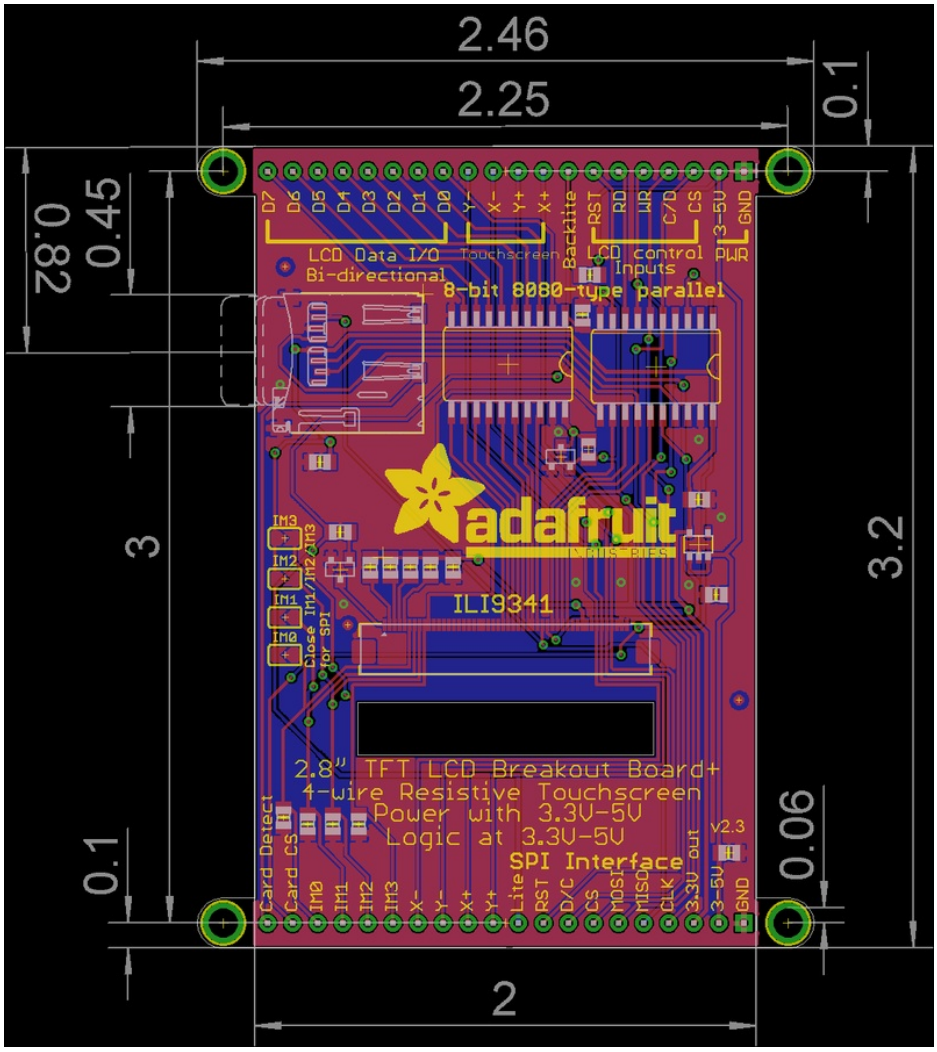
2.8" and 3.2" Resistive Touch Schematic

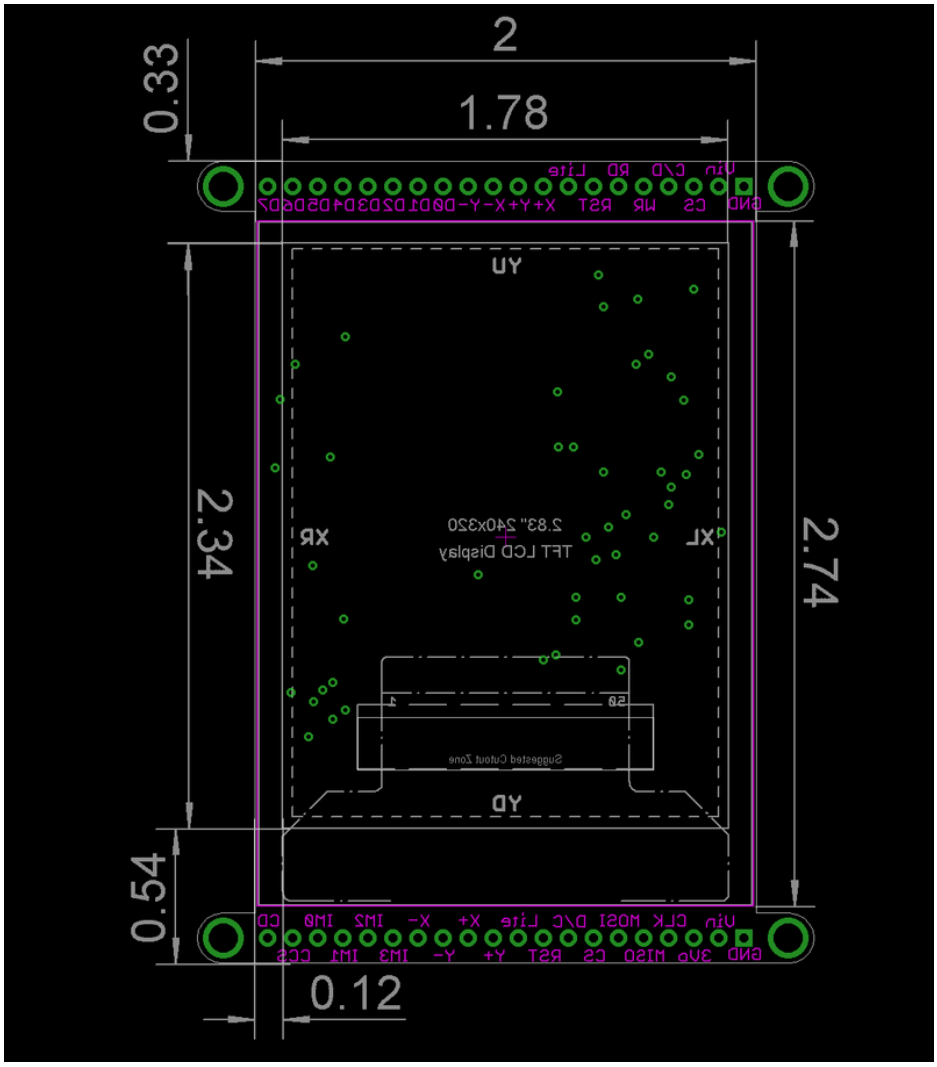


Capacitive Touch Schematic

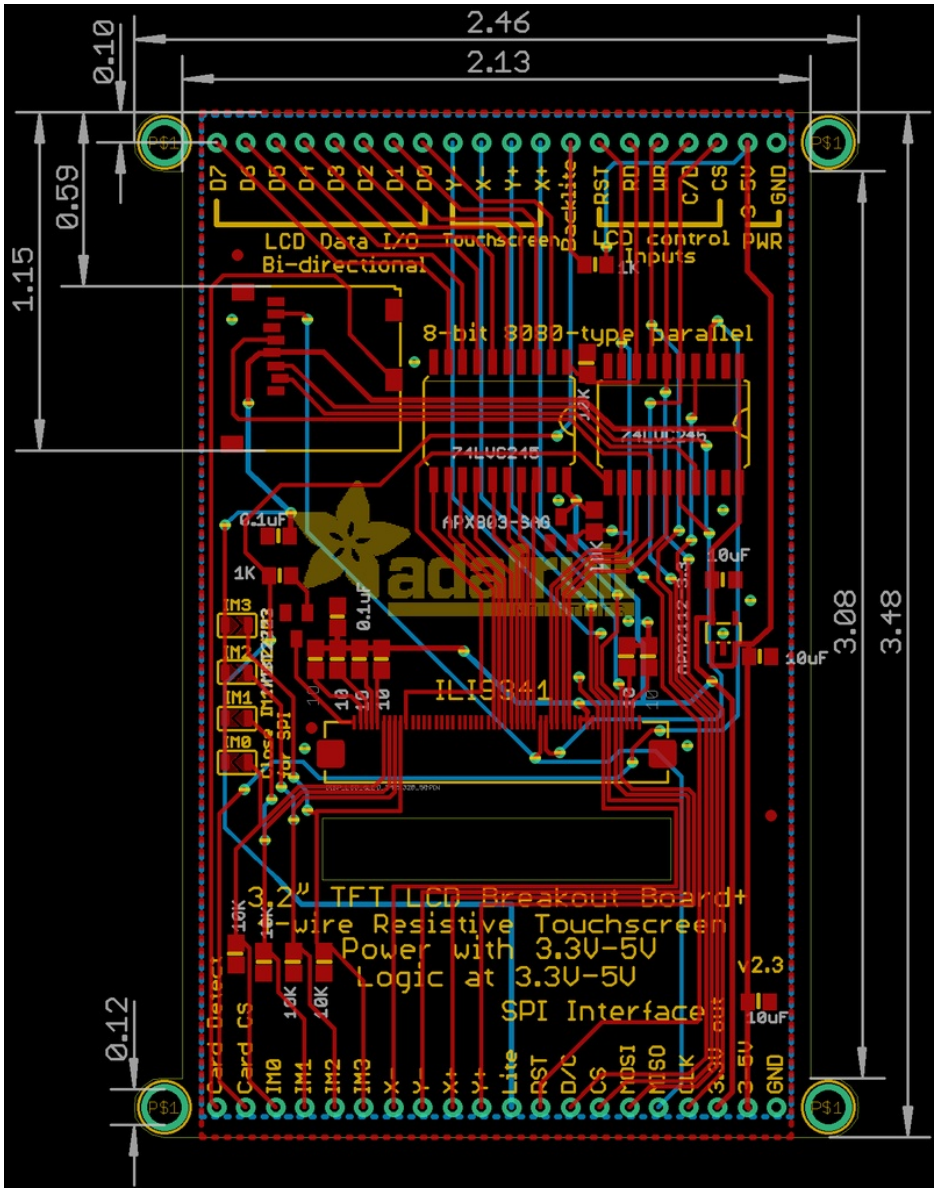


2.8" TFT Layout Diagram





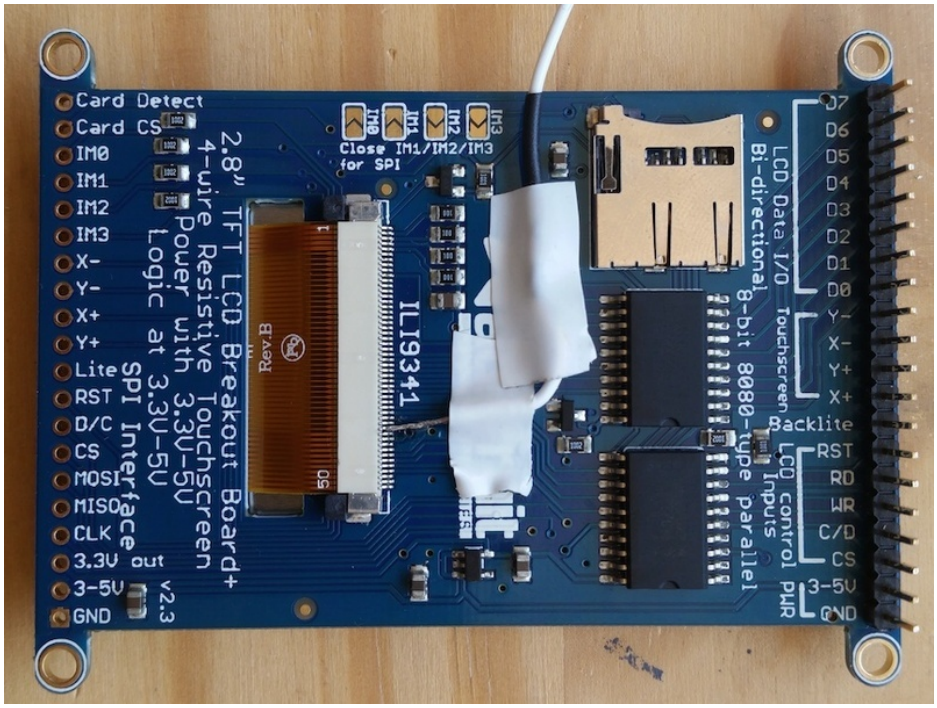
3.2" TFT Layout Diagram


2.46
2.13
0.10
0.59
1.15
3.08
3.48
0.12
LCD Data I/O
Bi-directional
Touchscreen
Backlite
LCD control
Inputs
PWR
8-bit 8080-type parallel
74LVC245
74LVC245
ILI9341
3.2" TFT LCD Breakout Board+
4-wire Resistive Touchscreen
Power with 3.3V-5V
Logic at 3.3V-5V
SPI Interface
v2.3
D7
D6
D5
D4
D3
D2
D1
D0
Y-
X-
Y+
X+
RST
RD
WR
C/D
CS
3.3V
5V
GND
Lite
RST
D/C
CS
MOSI
MISO
CLK
3.3V out
5V
GND
Card Detect
Card CS
IM0
IM1
IM2
IM3
0.1uF
1K
10K
10uF

F.A.Q.

- If I drive this display at very high speeds I get 'video tearing' effects, how can I synchronize the display refreshes?

  We don't break out the TE (tearing effect line) because we use these with small microcontrollers, but if you *do* need to synchronize you can solder to the TE pad on the TFT using fine silicone wire. (See this forum thread)



Display does not work on initial power but does work after a reset.

The display driver circuit needs a small amount of time to be ready after initial power. If your code tries to write to the display too soon, it may not be ready. It will work on reset since that typically does not cycle power. If you are having this issue, try adding a small amount of delay before trying to write to the display.

In Arduino, use **delay()** to add a few milliseconds before calling **tft.begin()**. Adjust the amount of delay as needed to see how little you can get away with for your specific setup.