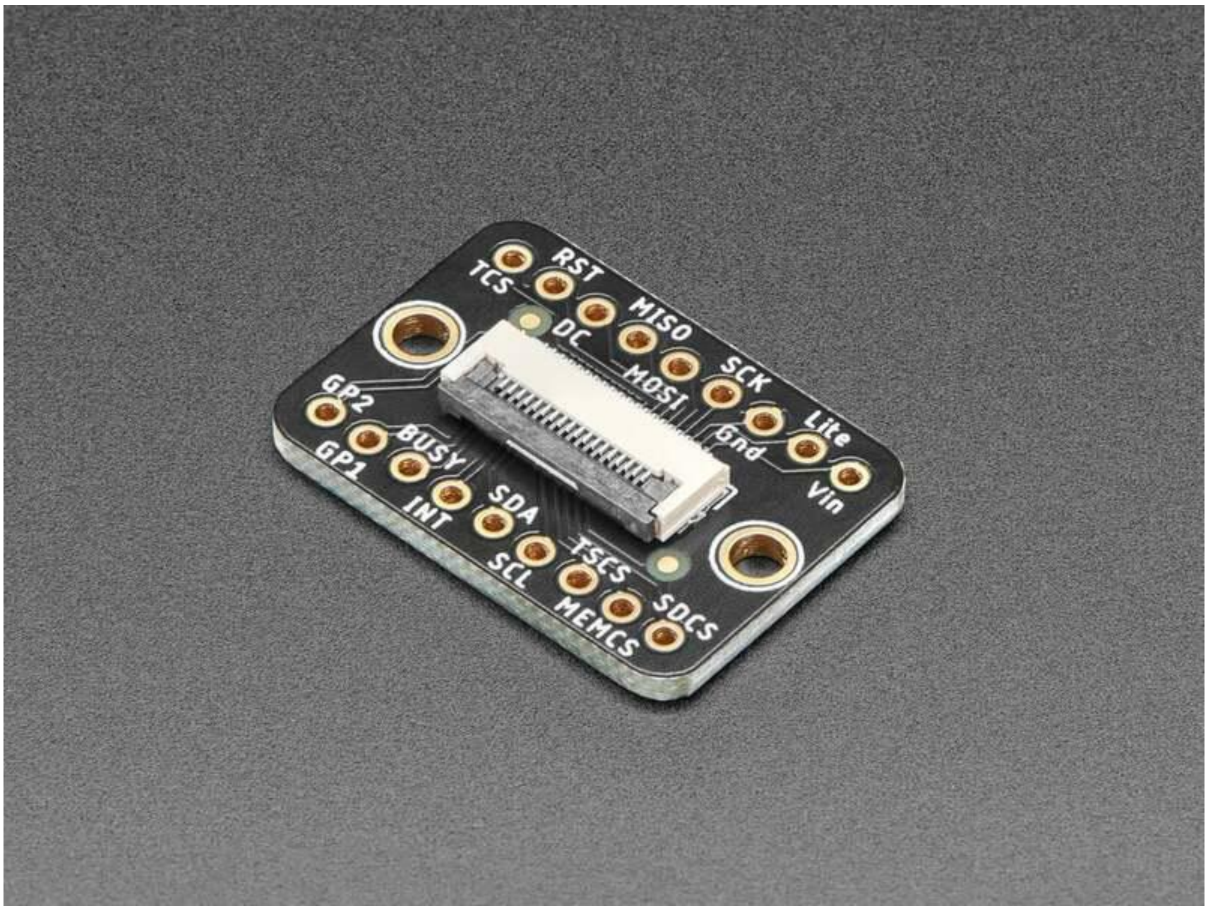
# Adafruit EYESPI Breakout Board

Created by Liz Clark

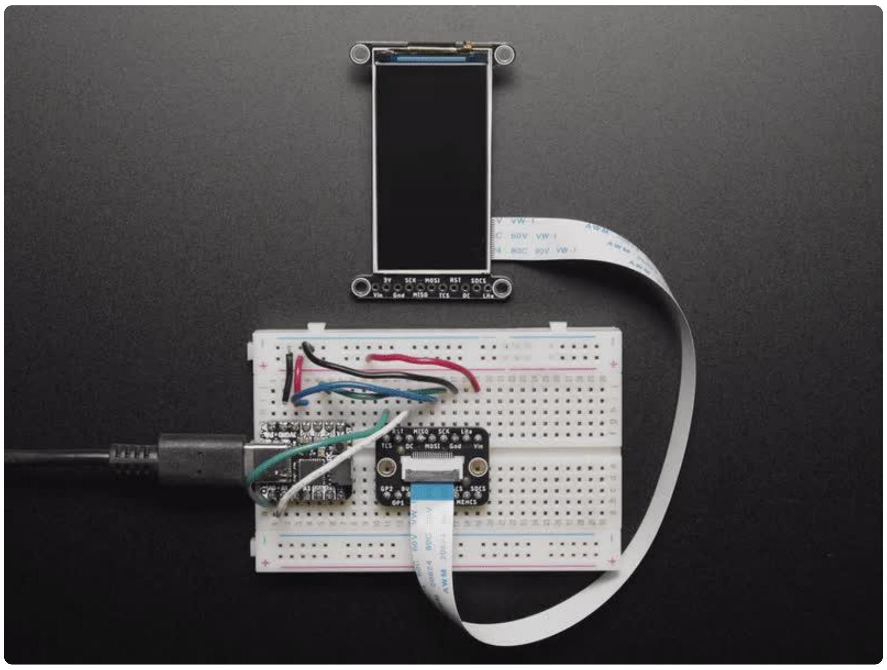


https://learn.adafruit.com/adafruit-eyespi-breakout-board

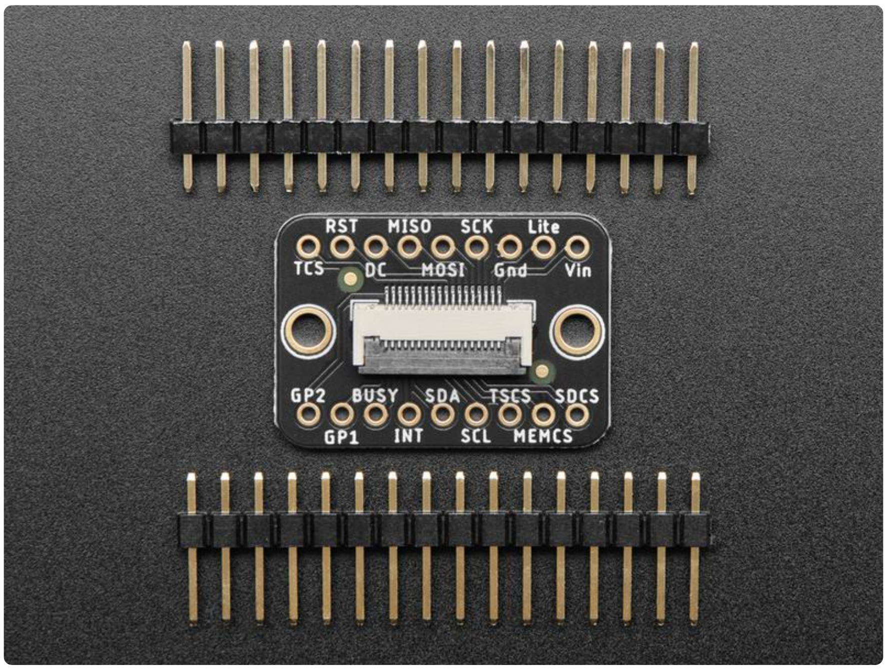
Last updated on 2023-08-29 04:53:04 PM EDT

# Table of Contents

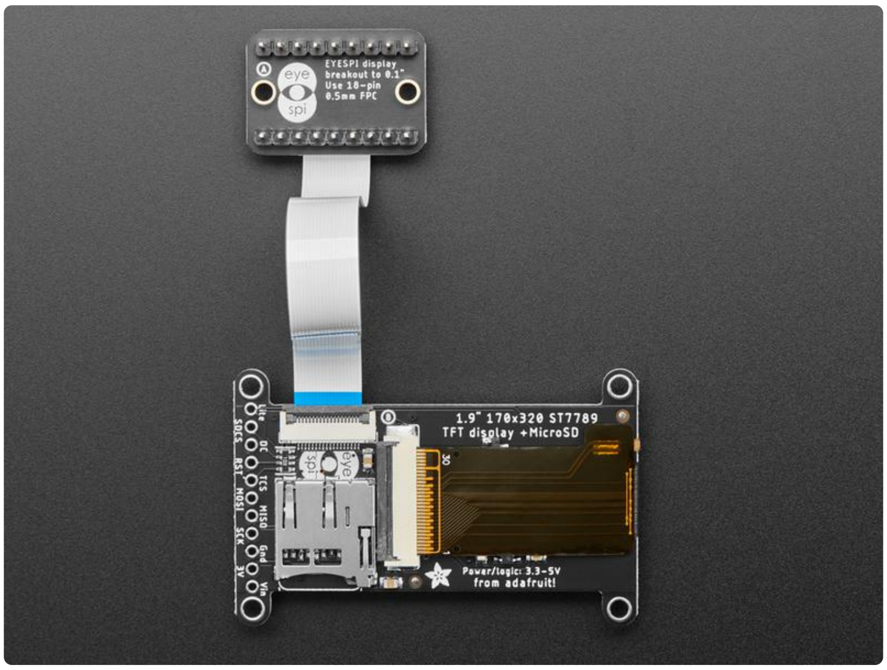
# Overview



Adafruit's most recent display breakouts have come with a new feature: an 18-pin "EYE SPI" standard FPC connector with flip-top connector (). This is intended to be a sort-of "STEMMA QT () for displays" - a way to quickly connect and extend display wiring that uses a lot of SPI pins. In this case, we need a lot of SPI pins, and we want to be able to use long distances, so the answer is an 18-pin 0.5mm pitch FPC connector.

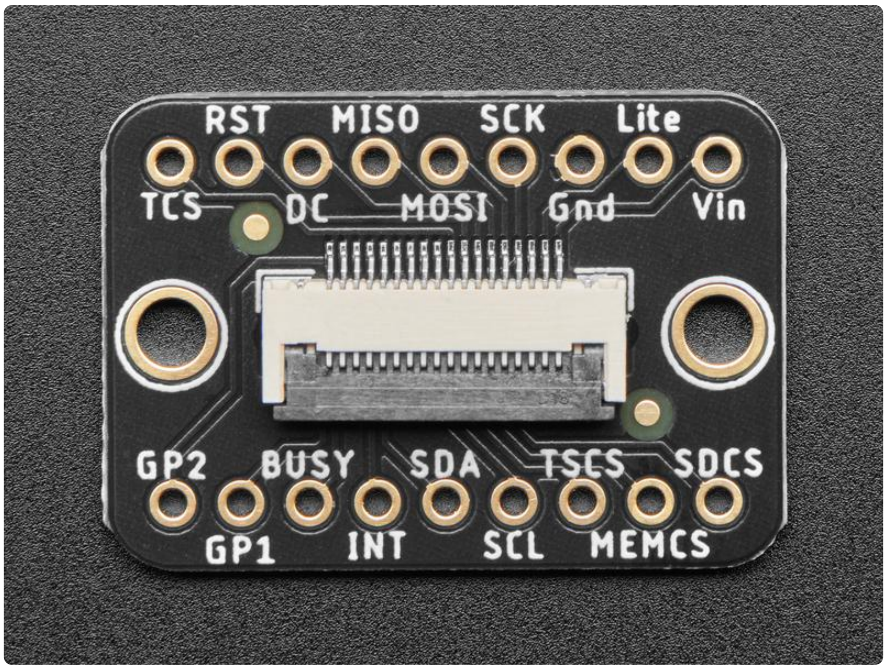


Now you can connect to the displays, but what goes on the other side of the FPC? Here is one possibility - a simple breakout board that brings all the GPIO to 0.1" spaced header, for breadboarding use. Each pin usage is labeled, and for most display purposes, you only need the left half of the display for power and SPI connectivity.


EYESPI display breakout to 0.1" Use 18-pin 0.5mm FPC
eye spi
1.9" 170x320 ST7789 TFT display +MicroSD
Power/logic: 3.3-5V from adafruit!

Don't forget: you'll also want an 18-pin EYESPI FPC cable ().

## Pinouts



The EYESPI breakout board breaks out all 18 pins of the display connector, for breadboarding use. This breakout is a passthrough for the display connector.

The connections you will actually use will vary from display to display, and often not all pins will be required.

## Power 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.
- Gnd - This is common ground for power and logic.

## I2C Logic Pins

- SCL - This is the I2C serial clock pin. Connect to the desired I2C clock pin on your microcontroller.
- SDA - This is the I2C serial data pin. Connect to the desired I2C data pin on your microcontroller.

## SPI Pins

- SCK - This is the SPI clock input pin.
- 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.
- 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).
- DC - This is the display SPI data/command selector 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.
- TCS - This is the TFT or eInk SPI chip select pin.

## GPIO Pins

- GP1 and GP2 - These are the GPIO pins.

## Chip Select Pins

- TSCS - This is the Touch Screen Chip Select pin.
- MEMCS - This is the Memory Chip Select. This pin is required for communicating with the onboard RAM chip.
- SDCS - This is the SD card chip select pin. This pin is required for communicating with the onboard SD card holder. You can leave this disconnected if you aren't going to access SD cards.

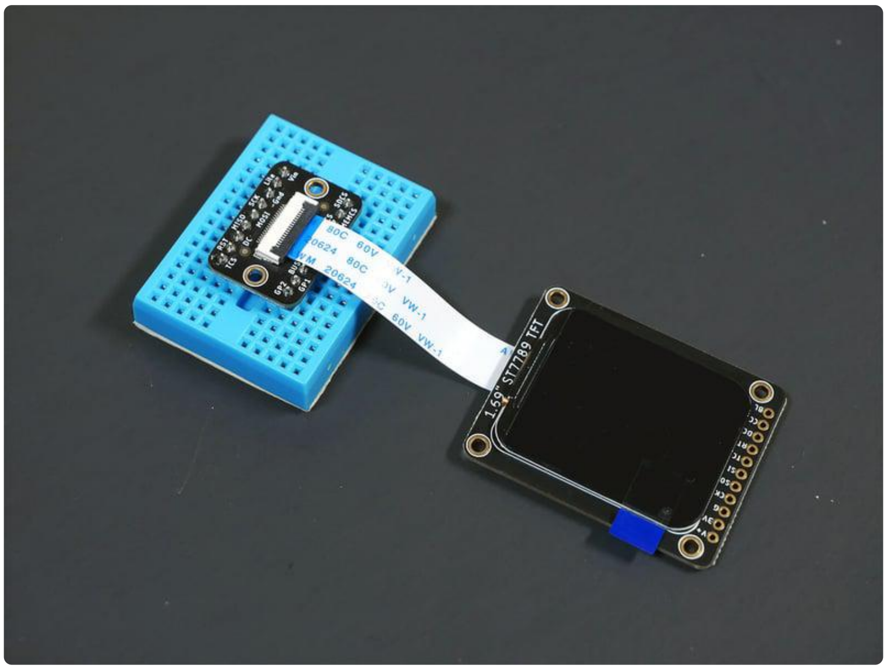
## Backlight Pin

- 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.

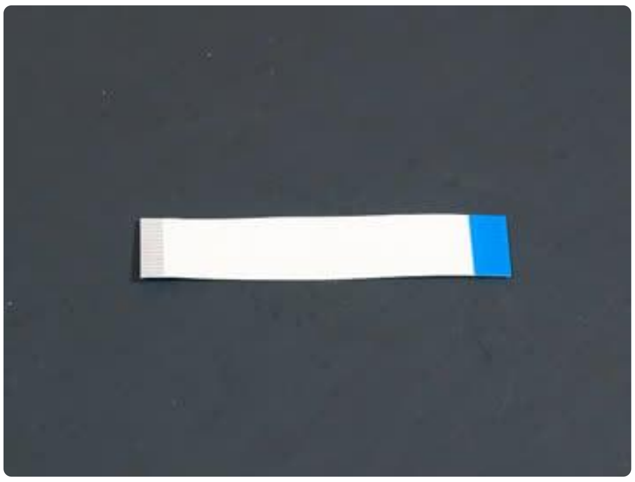
## Other Pins

- BUSY - This is the busy-detect pin used by eInk displays. It is optional, and if not connected, the code will wait an approximate number of seconds.
- INT - This is the capacitive touch interrupt pin. When a touch is detected, this pin goes low. This is only necessary when using a capacitive touch display.

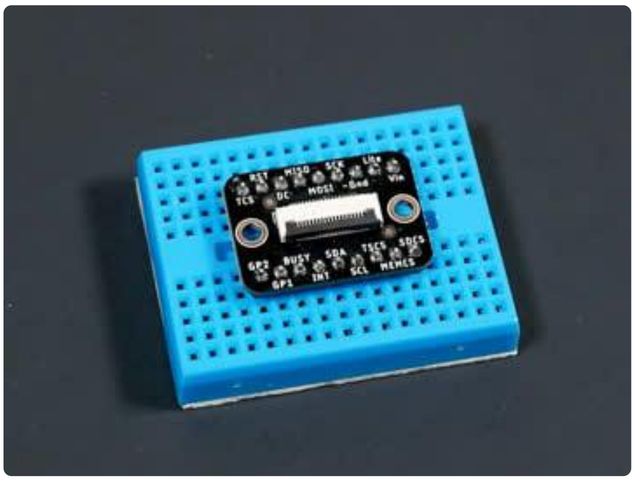
# Plugging in an EYESPI Cable



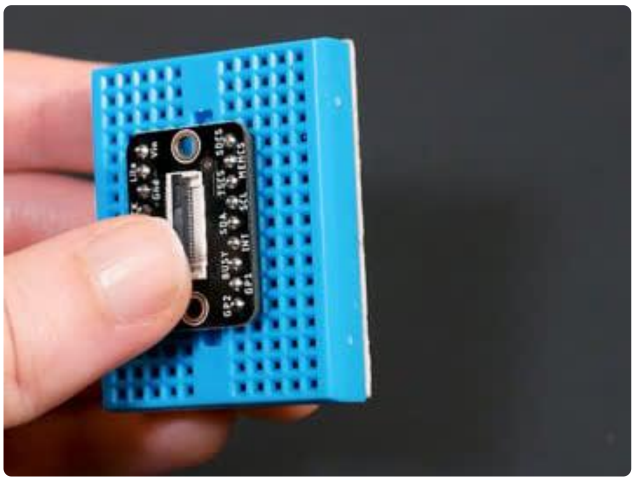
You can connect an EYESPI compatible display to the EYESPI breakout board using an EYESPI cable. An EYESPI cable is an 18 pin flexible PCB (FPC). The FPC can only be connected properly in one orientation, so be sure to follow the steps below to ensure that your display and breakout are plugged in properly.



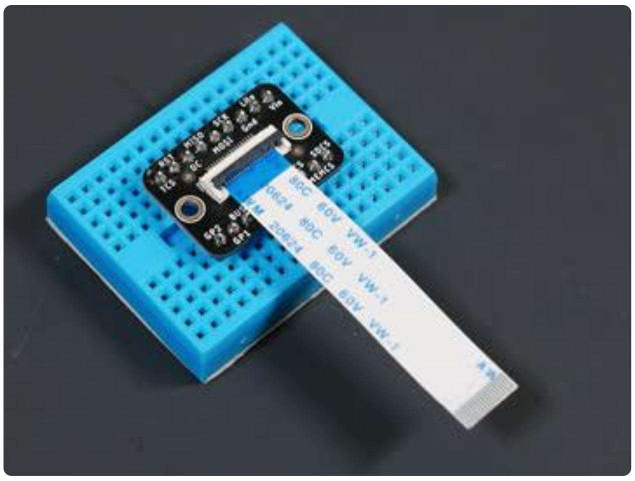
Each EYESPI cable has blue stripes on either end. On the other side of the cable, underneath the blue stripe, are the connector pins that make contact with the FPC connector pins on the display or breakout.



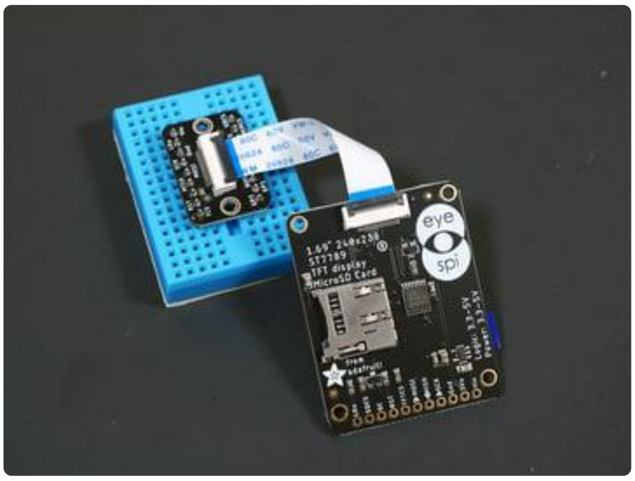
To begin inserting an EYESPI cable to an FPC connector, gently lift the FPC connector black latch up.



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.

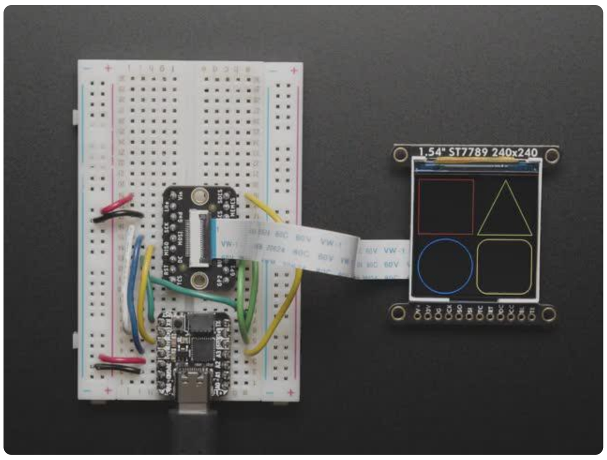


Repeat this process for the FPC connector on your display. Again, ensure that the blue stripe on either end of the cable is facing up.

## CircuitPython

Using the EYESPI breakout with CircuitPython involves wiring up the breakout to your CircuitPython-compatible microcontroller and plugging in your EYESPI-compatible display via an EYESPI cable. Then, you load the code and necessary libraries onto your microcontroller to see the graphics test on the display.

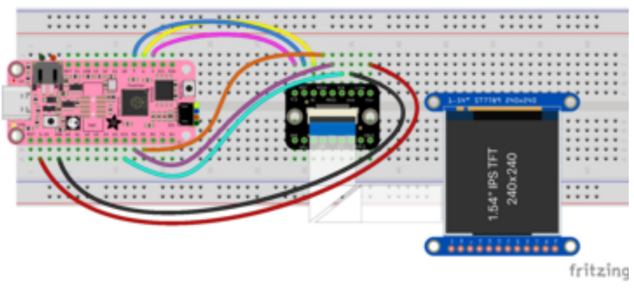
This page uses the 1.54" 240x240 ST7789 TFT display for demonstrating CircuitPython usage. You can use the same concepts to get going with any EYESPI-compatible display.



Adafruit 1.54" 240x240 Wide Angle TFT LCD Display with MicroSD
We've been looking for a display like this for a long time - it's only 1.5" diagonal but has a high density 220 ppi, 240x240 pixel display with full-angle viewing. It...
https://www.adafruit.com/product/3787

## Wiring

First, wire the EYESPI breakout to your CircuitPython-compatible microcontroller. The diagram below shows wiring up to a Feather RP2040. Then, connect your EYESPI-compatible display via an EYESPI cable to the breakout.



Feather 3.3V to breakout Vin
Feather GND to breakout Gnd
Feather SCK to breakout SCK
Feather MO to breakout MOSI
Feather MI to breakout MISO
Feather D9 to breakout RST
Feather D6 to breakout DC
Feather D5 to breakout TCS
Attach the TFT screen to the EYESPI breakout with an EYESPI cable as described on the Plugging in an EYESPI Cable () page.

## CircuitPython Usage

To use with CircuitPython, you need to first install the necessary libraries, and their dependencies, into the lib folder on your CIRCUITPY drive. Then you need to update code.py with the example script.

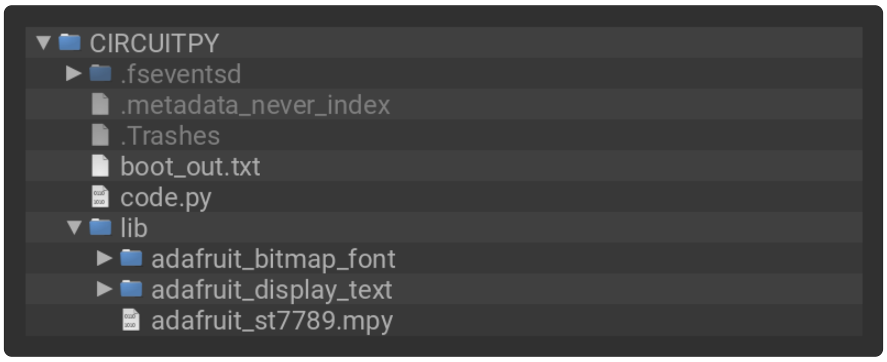
Thankfully, we can do this in one go. In the example below, click the Download Project Bundle button below to download the necessary libraries and the code.py file in a zip file. Extract the contents of the zip file.

Connect the microcontroller to your computer via a known-good USB power+data cable. The board shows up as a thumb drive named CIRCUITPY. Copy the entire lib folder, the tom-thumb.pcf font file, and the code.py file to your CIRCUITPY drive.

Your CIRCUITPY/lib folder should contain the following folders and files:

- /adafruit_bitmap_font
- /adafruit_display_text
- adafruit_st7789.mpy

Once you have copied over the necessary folders and files, your CIRCUITPY drive should resemble the following:



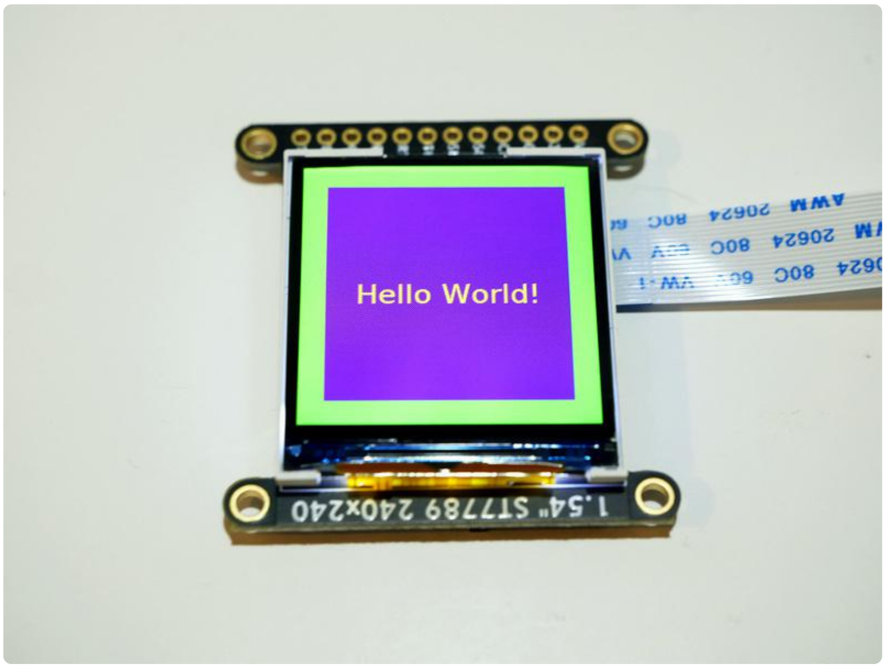
```
▼ CIRCUITPY
  ▶ .fseventsd
    .metadata_never_index
    .Trashes
    boot_out.txt
    code.py
  ▼ lib
    ▶ adafruit_bitmap_font
    ▶ adafruit_display_text
      adafruit_st7789.mpy
```

## Example Code

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

"""
This test will initialize the display using displayio and draw a solid green
background, a smaller purple rectangle, and some yellow text.
"""
import board
import terminalio
import displayio
from adafruit_display_text import label
from adafruit_st7789 import ST7789

# 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
)

display = ST7789(display_bus, width=240, height=240, rowstart=80)

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

```
display.show(splash)

color_bitmap = displayio.Bitmap(240, 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(200, 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(scale=2, x=50, 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
```

Once everything is copied over, on your display, you should see a green rectangle appear, followed by a smaller, inset purple rectangle, and finally, yellow text centered on the display saying, "Hello World!".



## CircuitPython Docs

CircuitPython Docs ()

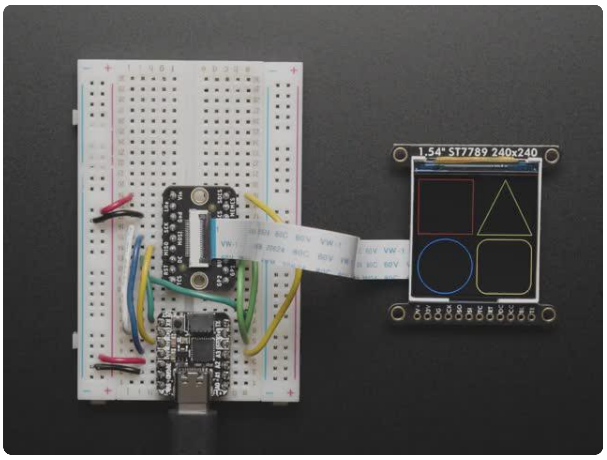
## Python

There's two ways you can use the EYESPI breakout board with a computer that has GPIO and Python thanks to Adafruit_Blinka, our CircuitPython-for-Python compatibility library (). You can utilize CPython graphics libraries, such as PIL (), when using the EYESPI breakout with Blinka.

Additionally, you can install a kernel module to add support for a supported TFT display that will make the console appear. This is cute because you can have any program print text or draw to the framebuffer (or, say, with pygame) and Linux will take care of displaying it for you. If you don't need the console or direct framebuffer access, please consider using the 'pure Python' technique instead as it is not as delicate.

Be aware that you can only choose to do one way at a time. If you choose the hard way, it will install the kernel driver, which will prevent you from doing it the easy way

Below is example wiring and code for an ST7789 TFT display using the Adafruit_CircuitPython_RGB_Display () library.

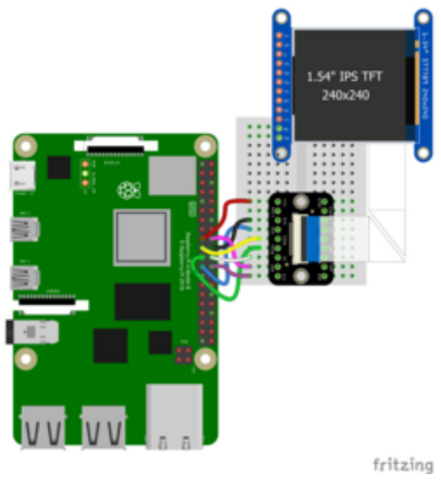


Adafruit 1.54" 240x240 Wide Angle TFT LCD Display with MicroSD
We've been looking for a display like this for a long time - it's only 1.5" diagonal but has a high density 220 ppi, 240x240 pixel display with full-angle viewing. It...
https://www.adafruit.com/product/3787

### Python Computer Wiring

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

Here's the Raspberry Pi wired to the EYESPI breakout board connected to an ST7789 1.54" 240x240 TFT Display:



Pi 3V to breakout VIN (red wire)
Pi GND to breakout GND (black wire)
Pi SLCK (GPIO 11) to breakout SCK (blue wire)
Pi MISO (GPIO 9) to breakout MISO (green wire)
Pi MOSI (GPIO 10) to breakout MOSI (yellow wire)
Pi CE0 (GPIO 8) to breakout TCS (purple wire)
Pi GPIO 25 to breakout DC (white wire)
Pi GPIO 24 to breakout RST (pink wire)
Attach the TFT screen to the EYESPI breakout with an EYESPI cable as described on the Plugging in an EYESPI Cable () page.

## Python Installation of the RGB Display Library

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

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. Make sure you aren't trying to use CircuitPython on Python 2.x, it isn't supported!

## Python Usage

Once you have the library `pip3` installed on your computer, copy or download the following example to your computer, and run the following, replacing code.py with whatever you named the file:

```
python3 code.py
```

## Example Code

```
# 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)
```

## Edit the Code for the ST7789 Display

In the code, comment out the ILI9341 `disp` object and replace it with an ST7789 `disp` object:

```
'''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,
)'''
disp = st7789.ST7789(
    spi,
    height=240,
    y_offset=80,
    rotation=180,
    cs=cs_pin,
    dc=dc_pin,
    rst=reset_pin,
    baudrate=BAUDRATE,
)
```

The code begins by instantiating your display over SPI. Then, a green rectangle is drawn, followed by a smaller purple rectangle. The text `"Hello World"` is displayed over both of the rectangles.

## Kernel Module Install

You can install a kernel module using one of Adafruit's Raspberry-Pi-Installer-Scripts () to view your Raspberry Pi's terminal or desktop on a display connected to the EYESPI breakout board. This can be handy for projects such as cyberdecks or headless setups. Please proceed with caution though, as it can be tricky to setup and you may find that a 'pure Python' method with Blinka and Pillow (detailed above) can answer your Raspberry Pi display needs.

The following will detail an example of showing the terminal via an ST7789 display. If you want to run it on a different display, please reference this guide on PiTFT (), which has terminal inputs for all of the supported displays.

PiTFT Display for Raspberry Pi Learn Guide - Installation Page

## Prepare the Pi

After wiring your ST7789 display to the Raspberry Pi with the EYESPI breakout as detailed above (), its a good idea to get your Pi completely updated and upgraded. We assume you have burned an SD card and can log into the console to install stuff.

To update and upgrade, run:

```
sudo apt update -y
sudo apt-get update -y
sudo apt-get upgrade -y
```

Then, reboot the Pi with:

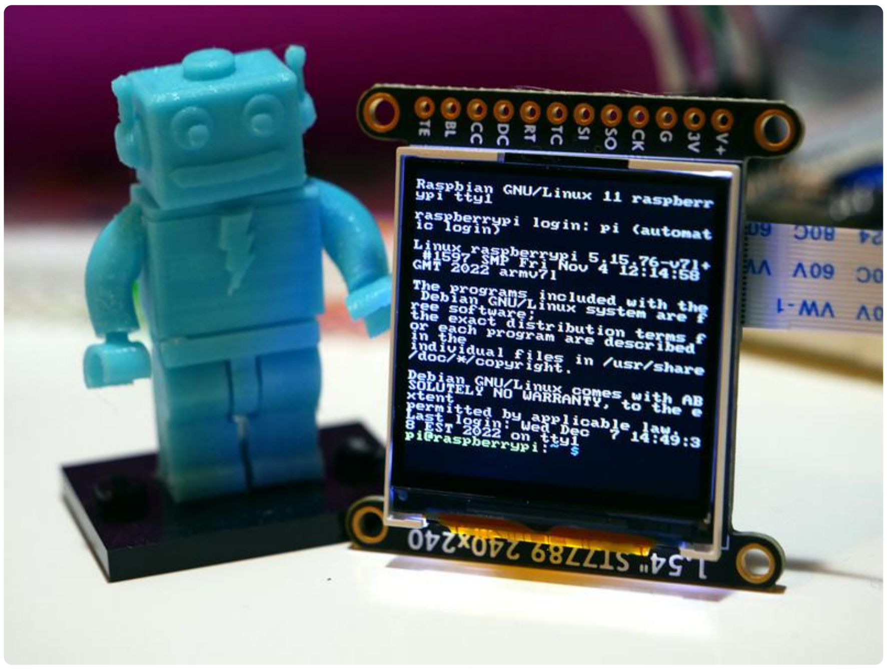
```
sudo reboot
```

Then, run the following at the terminal:

```
cd ~
sudo pip3 install --upgrade adafruit-python-shell click
sudo apt-get install -y git
git clone https://github.com/adafruit/Raspberry-Pi-Installer-Scripts.git
cd Raspberry-Pi-Installer-Scripts
sudo python3 adafruit-pitft.py --display=st7789_240x240 --rotation=0 --install-
type=console
```

When you get asked to reboot, reboot!

After rebooting, you'll see the terminal on the ST7789 TFT display.



```
Raspbian GNU/Linux 11 raspberr
ypi tty1

raspberrypi login: pi (automat
ic login)

Linux raspberrypi 5.15.76-v7l+
 #1597 SMP Fri Nov 4 12:14:58
GMT 2022 armv7l

The programs included with the
 Debian GNU/Linux system are f
ree software;
the exact distribution terms f
or each program are described
in the
individual files in /usr/share
/doc/*/copyright.

Debian GNU/Linux comes with AB
SOLUTELY NO WARRANTY, to the e
xtent
permitted by applicable law.
Last login: Wed Dec  7 14:49:3
8 EST 2022 on tty1
pi@raspberrypi:~ $
```

## Uninstall the Kernel Module

If you decide that you no longer want to show the terminal on the ST7789 display, you can run the following at the terminal:

```
cd ~
cd Raspberry-Pi-Installer-Scripts
sudo python3 adafruit-pitft.py --install-type=uninstall
```

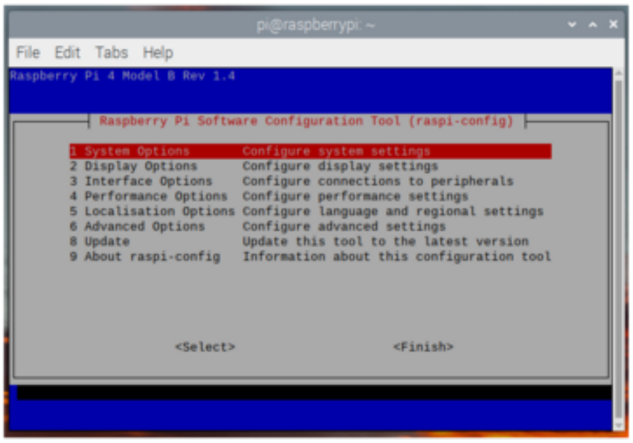
When you get asked to reboot, reboot!

After rebooting, you'll be brought to the terminal on your HDMI connected display. If you want to get back to the desktop, enter `startx` into the terminal.

```
startx
```

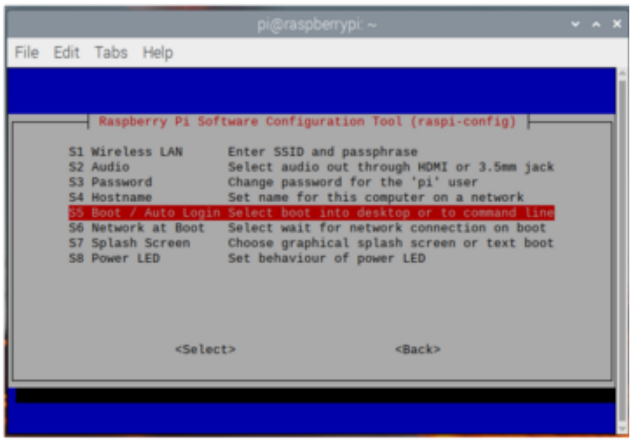
To permanently boot to the desktop going forward, enter the `raspi-config` tool via the terminal with:

```
sudo raspi-config
```



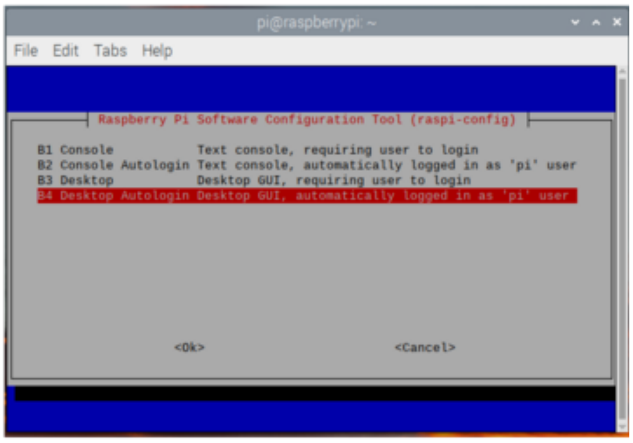
```
Raspberry Pi 4 Model B Rev 1.4

Raspberry Pi Software Configuration Tool (raspi-config)

1 System Options       Configure system settings
2 Display Options      Configure display settings
3 Interface Options    Configure connections to peripherals
4 Performance Options  Configure performance settings
5 Localisation Options Configure language and regional settings
6 Advanced Options     Configure advanced settings
8 Update               Update this tool to the latest version
9 About raspi-config   Information about this configuration tool

         <Select>                  <Finish>
```

Select 1 System Options in the menu.



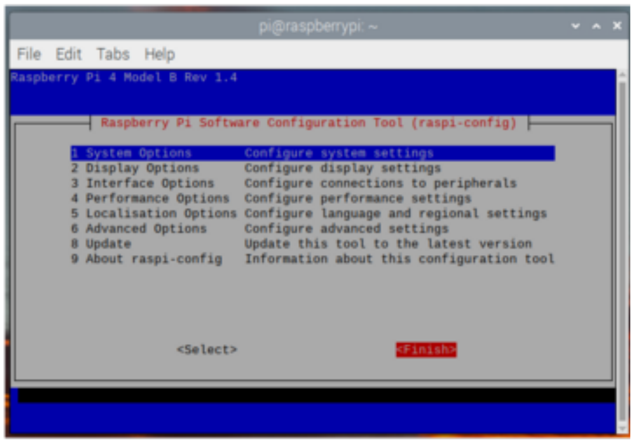
```
Raspberry Pi Software Configuration Tool (raspi-config)

S1 Wireless LAN      Enter SSID and passphrase
S2 Audio             Select audio out through HDMI or 3.5mm jack
S3 Password          Change password for the 'pi' user
S4 Hostname          Set name for this computer on a network
S5 Boot / Auto Login Select boot into desktop or to command line
S6 Network at Boot   Select wait for network connection on boot
S7 Splash Screen     Choose graphical splash screen or text boot
S8 Power LED         Set behaviour of power LED

         <Select>                    <Back>
```

On the next screen, select S5 Boot / Auto Login.

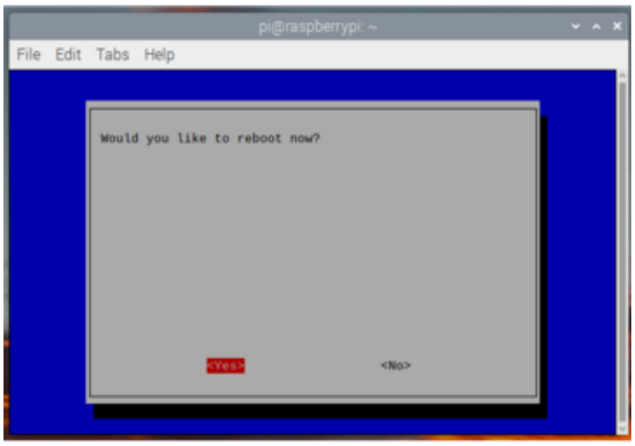


```
Raspberry Pi Software Configuration Tool (raspi-config)

B1 Console           Text console, requiring user to login
B2 Console Autologin Text console, automatically logged in as 'pi' user
B3 Desktop           Desktop GUI, requiring user to login
B4 Desktop Autologin Desktop GUI, automatically logged in as 'pi' user

         <Ok>                        <Cancel>
```

Finally, select B4 Desktop Autologin to return to the default desktop login mode.



```
Raspberry Pi 4 Model B Rev 1.4

Raspberry Pi Software Configuration Tool (raspi-config)

1 System Options       Configure system settings
2 Display Options      Configure display settings
3 Interface Options    Configure connections to peripherals
4 Performance Options  Configure performance settings
5 Localisation Options Configure language and regional settings
6 Advanced Options     Configure advanced settings
8 Update               Update this tool to the latest version
9 About raspi-config   Information about this configuration tool

          <Select>                     <Finish>
```



```
Would you like to reboot now?

          <Yes>                <No>
```

Exit the `raspi-config` by selecting Finish. When prompted to reboot, select Yes.

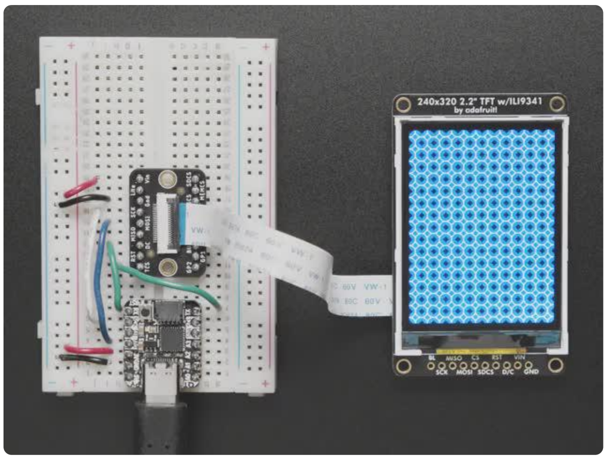
## Python Docs

Python Docs ()

## Arduino

Using the EYESPI breakout with Arduino involves wiring up the breakout to your Arduino-compatible microcontroller, plugging in your EYESPI compatible screen via the EYESPI cable, installing the library for your display type and running the provided example code.

Below is example wiring and code for an ILI9341 TFT display using the Adafruit_ILI9341 () library.

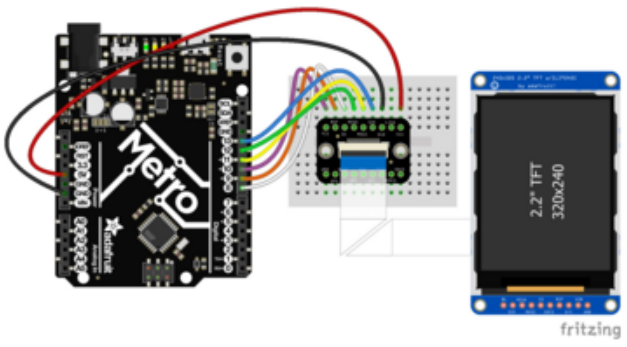


2.2" 18-bit color TFT LCD display with microSD card breakout
This lovely little display breakout is the best way to add a small, colorful, and bright display to any project. Since the display uses 4-wire SPI to communicate and has its own...
https://www.adafruit.com/product/1480

## Wiring

Wire as shown for a 5V board like an Uno. If you are using a 3V board, like an Adafruit Feather, wire the board's 3V pin to the EYESPI breakout VIN.

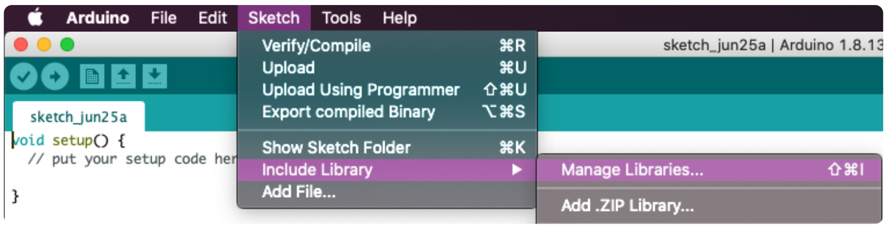
Here is an Adafruit Metro wired up to the EYESPI breakout with an ILI9341 240x320 2.2" TFT Display:



Board 5V to breakout VIN (red wire)
Board GND to breakout GND (black wire)
Board pin 13 to breakout SCK (blue wire)
Board pin 12 to breakout MISO (green wire)
Board pin 11 to breakout MOSI (yellow wire)
Board pin 10 to breakout TCS (purple wire)
Board pin 9 to breakout RST (orange wire)
Board pin 8 to breakout DC (white wire)
Attach the TFT screen to the EYESPI breakout with an EYESPI cable as described on the Plugging in an EYESPI Cable () page.

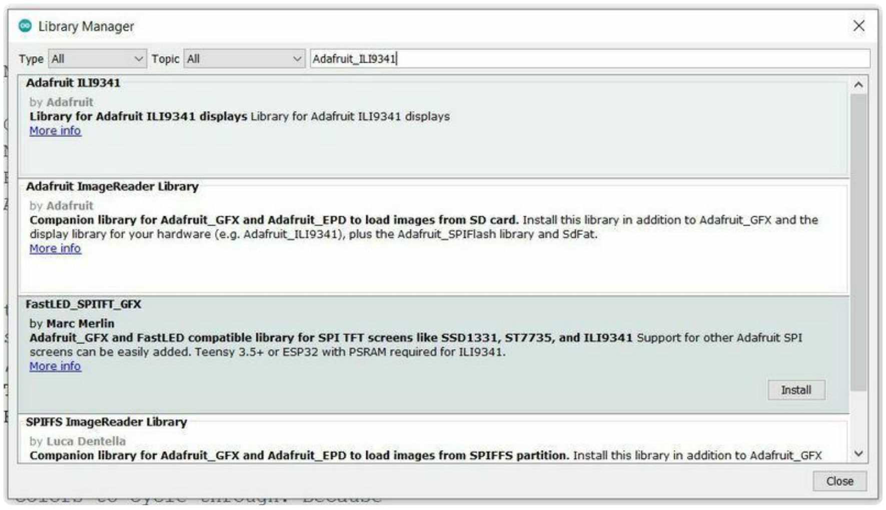
## Library Installation

You can install the Adafruit ILI9341 library for Arduino using the Library Manager in the Arduino IDE.



```
void setup() {
  // put your setup code her
}
```

Click the Manage Libraries ... menu item, search for Adafruit ILI9341, and select the Adafruit_ILI9341 library:



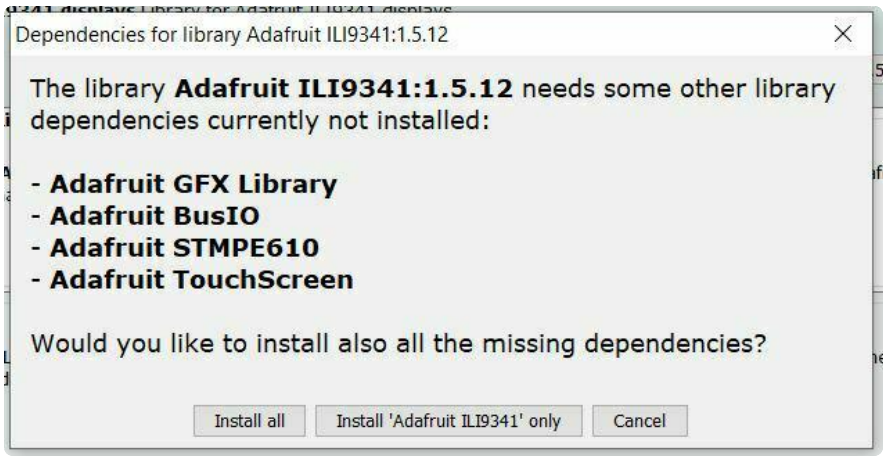
**Adafruit ILI9341**
by Adafruit
**Library for Adafruit ILI9341 displays** Library for Adafruit ILI9341 displays
More info

**Adafruit ImageReader Library**
by Adafruit
**Companion library for Adafruit_GFX and Adafruit_EPD to load images from SD card.** Install this library in addition to Adafruit_GFX and the display library for your hardware (e.g. Adafruit_ILI9341), plus the Adafruit_SPIFlash library and SdFat.
More info

**FastLED_SPITFT_GFX**
by **Marc Merlin**
**Adafruit_GFX and FastLED compatible library for SPI TFT screens like SSD1331, ST7735, and ILI9341** Support for other Adafruit SPI screens can be easily added. Teensy 3.5+ or ESP32 with PSRAM required for ILI9341.
More info
Install

**SPIFFS ImageReader Library**
by Luca Dentella
**Companion library for Adafruit_GFX and Adafruit_EPD to load images from SPIFFS partition.** Install this library in addition to Adafruit_GFX

If asked about dependencies, click "Install all".


Dependencies for library Adafruit ILI9341:1.5.12

The library **Adafruit ILI9341:1.5.12** needs some other library dependencies currently not installed:

- **Adafruit GFX Library**
- **Adafruit BusIO**
- **Adafruit STMPE610**
- **Adafruit TouchScreen**

Would you like to install also all the missing dependencies?

If the "Dependencies" window does not come up, then you already have the dependencies installed.

If the dependencies are already installed, you must make sure you update them through the Arduino Library Manager before loading the example!

## Example Code

```
// SPDX-FileCopyrightText: 2022 Phillip Burgess for Adafruit Industries
//
// SPDX-License-Identifier: MIT

// Graphics example for EYESPI-capable color displays. This code:
// - Functions as a "Hello World" to verify that microcontroller and screen
//   are communicating.
// - Demonstrates most of the drawing commands of the Adafruit_GFX library.
// - Showcases some techniques that might not be obvious or that aren't
//   built-in but can be handled with a little extra code.
// It DOES NOT:
// - Support all Adafruit screens, ONLY EYESPI products at the time this was
//   written! But it's easily adapted by looking at other examples.
// - Demonstrate the Adafruit_GFX_Button class, as that's unique to
//   touch-capable displays. Again, other examples may cover this topic.
// This sketch is long, but a lot of it is comments to explain each step. You
// can copy just the parts you need as a starting point for your own projects,
// and strip comments once understood.

// CONSTANTS, HEADERS and GLOBAL VARIABLES ---------------------------------

// *** EDIT THIS VALUE TO MATCH THE ADAFRUIT PRODUCT ID FOR YOUR DISPLAY: ***
#define SCREEN_PRODUCT_ID 5393
// You can find the product ID several ways:
// - "PID" accompanies each line-item on your receipt or order details page.
// - Visit adafruit.com and search for EYESPI displays. On product pages,
//   PID is shown just below product title, and is at the end of URLs.
// - Check the comments in setup() later that reference various screens.

// **** EDIT PINS TO MATCH YOUR WIRING ****
#define TFT_CS  10 // To display chip-select pin
#define TFT_RST  9 // To display reset pin
#define TFT_DC   8 // To display data/command pin
// For the remaining pins, this code assumes display is wired to hardware SPI
// on the dev board's primary SPI interface. The display libraries can support
// secondary SPI (if present) or bitbang (software) SPI, but that's not
// demonstrated here. See other examples for more varied interfacing options.

#include <Adafruit_GFX.h>             // Core graphics library
#include <Fonts/FreeSansBold18pt7b.h> // A custom font
#if (SCREEN_PRODUCT_ID == 1480) || (SCREEN_PRODUCT_ID == 2090)
#include <Adafruit_ILI9341.h>         // Library for ILI9341-based screens
Adafruit_ILI9341 display(TFT_CS, TFT_DC, TFT_RST);
#else
#include <Adafruit_ST7789.h>          // Library for ST7789-based screens
Adafruit_ST7789  display(TFT_CS, TFT_DC, TFT_RST);
#endif

#define PAUSE 3000  // Delay (millisecondss) between examples
uint8_t rotate = 0; // Current screen orientation (0-3)

// setup() RUNS ONCE AT PROGRAM STARTUP ------------------------------------

void setup() {
  // Initialize display hardware
#if (SCREEN_PRODUCT_ID == 5393)   // 1.47" 320x172 round-rect TFT
#define CORNER_RADIUS 22
  display.init(172, 320);
#elif (SCREEN_PRODUCT_ID == 3787) // 1.54" 240x240 TFT
  display.init(240, 240);
#elif (SCREEN_PRODUCT_ID == 5206) // 1.69" 280x240 round-rect TFT
#define CORNER_RADIUS 43
  display.init(240, 280);
#elif (SCREEN_PRODUCT_ID == 5394) // 1.9" 320x170 TFT
```

```
  display.init(170, 320);
#else                            // All ILI9341 TFTs (320x240)
  display.begin();
#endif
#if !defined(CORNER_RADIUS)
#define CORNER_RADIUS 0
#endif

  // OPTIONAL: default TFT SPI speed is fairly conservative, you can try
  // overriding here for faster screen updates. Actual SPI speed may be less
  // depending on microcontroller's capabilities. Max reliable speed also
  // depends on wiring length and tidyness.
  //display.setSPISpeed(40000000);
}

// MAIN LOOP, REPEATS FOREVER -----------------------------------------------

void loop() {
  // Each of these functions demonstrates a different Adafruit_GFX concept:
  show_shapes();
  show_charts();
  show_basic_text();
  show_char_map();
  show_custom_text();
  show_bitmap();
#if !defined(AVR)
  // The full set of examples (plus the custom font) won't fit on an 8-bit
  // Arduino, something's got to go. You can try out this one IF the other
  // examples are disabled instead.
  show_canvas();
#endif

  if (++rotate > 3) rotate = 0; // Cycle through screen rotations 0-3
  display.setRotation(rotate);  // Takes effect on next drawing command
}

// BASIC SHAPES EXAMPLE -----------------------------------------------------

void show_shapes() {
  // Draw outlined and filled shapes. This demonstrates:
  // - Enclosed shapes supported by GFX (points & lines are shown later).
  // - Adapting to different-sized displays, and to rounded corners.

  const int16_t cx = display.width() / 2;  // Center of screen =
  const int16_t cy = display.height() / 2; // half of width, height
  int16_t minor = min(cx, cy);             // Lesser of half width or height
  // Shapes will be drawn in a square region centered on the screen. But one
  // particular screen -- rounded 240x280 ST7789 -- has VERY rounded corners
  // that would clip a couple of shapes if drawn full size. If using that
  // screen type, reduce area by a few pixels to avoid drawing in corners.
  if (CORNER_RADIUS > 40) minor -= 4;
  const uint8_t pad = 5;                   // Space between shapes is 2X this
  const int16_t size = minor - pad;        // Shapes are this width & height
  const int16_t half = size / 2;           // 1/2 of shape size

  display.fillScreen(0); // Start by clearing the screen; color 0 = black

  // Draw outline version of basic shapes: rectangle, triangle, circle and
  // rounded rectangle in different colors. Rather than hardcoded numbers
  // for position and size, some arithmetic helps adapt to screen dimensions.
  display.drawRect(cx - minor, cy - minor, size, size, 0xF800);
  display.drawTriangle(cx + pad, cy - pad, cx + pad + half, cy - minor,
                       cx + minor - 1, cy - pad, 0x07E0);
  display.drawCircle(cx - pad - half, cy + pad + half, half, 0x001F);
  display.drawRoundRect(cx + pad, cy + pad, size, size, size / 5, 0xFFE0);
  delay(PAUSE);

  // Draw same shapes, same positions, but filled this time.
  display.fillRect(cx - minor, cy - minor, size, size, 0xF800);
  display.fillTriangle(cx + pad, cy - pad, cx + pad + half, cy - minor,
                       cx + minor - 1, cy - pad, 0x07E0);
  display.fillCircle(cx - pad - half, cy + pad + half, half, 0x001F);
  display.fillRoundRect(cx + pad, cy + pad, size, size, size / 5, 0xFFE0);
  delay(PAUSE);
} // END SHAPE EXAMPLE

// CHART EXAMPLES ----------------------------------------------------------

void show_charts() {
  // Draw some graphs and charts. GFX library doesn't handle these as native
  // object types, but it only takes a little code to build them from simple
  // shapes. This demonstrates:
  // - Drawing points and horizontal, vertical and arbitrary lines.
  // - Adapting to different-sized displays.
  // - Graphics being clipped off edge.
  // - Use of negative values to draw shapes "backward" from an anchor point.
  // - C technique for finding array size at runtime (vs hardcoding).

  display.fillScreen(0);  // Clear screen

  const int16_t cx = display.width() / 2;  // Center of screen =
  const int16_t cy = display.height() / 2; // half of width, height
  const int16_t minor = min(cx, cy);       // Lesser of half width or height
  const int16_t major = max(cx, cy);       // Greater of half width or height

  // Let's start with a relatively simple sine wave graph with axes.
  // Draw graph axes centered on screen. drawFastHLine() and drawFastVLine()
  // need fewer arguments than normal 2-point line drawing shown later.
  display.drawFastHLine(0, cy, display.width(), 0x0210);  // Dark blue
  display.drawFastVLine(cx, 0, display.height(), 0x0210);

  // Then draw some tick marks along the axes. To keep this code simple,
  // these aren't to any particular scale, but a real program may want that.
  // The loop here draws them from the center outward and pays no mind
  // whether the screen is rectangular; any ticks that go off-screen will
  // be clipped by the library.
  for (uint8_t i=1; i<=10; i++) {
    // The Arduino map() function scales an input value (e.g. "i") from an
    // input range (0-10 here) to an output range (0 to major-1 here).
    // Very handy for making graphics adjust to different screens!
    int16_t n = map(i, 0, 10, 0, major - 1); // Tick offset relative to center point
    display.drawFastVLine(cx - n, cy - 5, 11, 0x210);
    display.drawFastVLine(cx + n, cy - 5, 11, 0x210);
    display.drawFastHLine(cx - 5, cy - n, 11, 0x210);
    display.drawFastHLine(cx - 5, cy + n, 11, 0x210);
  }

  // Then draw sine wave over this using GFX drawPixel() function.
  for (int16_t x=0; x<display.width(); x++) { // Each column of screen...
    // Note the inverted Y axis here (cy-value rather than cy+value)
    // because GFX, like most graphics libraries, has +Y heading down,
    // vs. classic Cartesian coords which have +Y heading up.
    int16_t y = cy - (int16_t)(sin((x - cx) * 0.05) * (float)minor * 0.5);
    display.drawPixel(x, y, 0xFFFF);
  }

  delay(PAUSE);

  // Next, let's draw some charts...
  // NOTE: some other examples in this code take extra steps to avoid placing
  // anything off in the rounded corners of certain displays. The charts do
  // not. It's *possible* but would introduce a lot of complexity into code
  // that's trying to show the basics. We'll leave the clipped charts here as
  // a teachable moment: not all content suits all displays.

  // A list of data to plot. These are Y values only; X assumed equidistant.
  const uint8_t data[] = { 31, 42, 36, 58, 67, 88 };       // Percentages, 0-100
  const uint8_t num_points = sizeof data / sizeof data[0]; // Length of data[] list
  display.fillScreen(0);  // Clear screen
  display.setFont();      // Use default (built-in) font
  display.setTextSize(2); // and 2X size for chart label

  // Chart label is centered manually; 144 is the width in pixels of
  // "Widget Sales" at 2X scale (12 chars * 6 px * 2 = 144). A later example
  // shows automated centering based on string.
  display.setCursor((display.width() - 144) / 2, 0);
  display.print(F("Widget Sales")); // F("string") is in program memory, not RAM
  // The chart-drawing code is then written to skip the top 20 rows where
  // this label is located.

  // First, a line chart, connecting the values point-to-point:

  // Draw a grid of lines to provide scale & an interesting background.
  for (uint8_t i=0; i<11; i++) {
    int16_t x = map(i, 0, 10, 0, display.width() - 1);   // Scale grid X to screen
    display.drawFastVLine(x, 20, display.height(), 0x001F);
    int16_t y = map(i, 0, 10, 20, display.height() - 1); // Scale grid Y to screen
    display.drawFastHLine(0, y, display.width(), 0x001F);
  }
  // And then draw lines connecting data points. Load up the first point...
  int16_t prev_x = 0;
  int16_t prev_y = map(data[0], 0, 100, display.height() - 1, 20);
  // Then connect lines to each subsequent point...
  for (uint8_t i=1; i<num_points; i++) {
    int16_t new_x = map(i, 0, num_points - 1, 0, display.width() - 1);
    int16_t new_y = map(data[i], 0, 100, display.height() - 1, 20);
    display.drawLine(prev_x, prev_y, new_x, new_y, 0x07FF);
    prev_x = new_x;
    prev_y = new_y;
  }
  // For visual interest, let's add a circle around each data point. This is
  // done in a second pass so the circles are always drawn "on top" of lines.
  for (uint8_t i=0; i<num_points; i++) {
    int16_t x = map(i, 0, num_points - 1, 0, display.width() - 1);
    int16_t y = map(data[i], 0, 100, display.height() - 1, 20);
    display.drawCircle(x, y, 5, 0xFFFF);
  }

  delay(PAUSE);

  // Then a bar chart of the same data...

  // Erase the old chart but keep the label at top.
  display.fillRect(0, 20, display.width(), display.height() - 20, 0);

  // Just draw the Y axis lines; bar chart doesn't really need X lines.
  for (uint8_t i=0; i<11; i++) {
    int16_t y = map(i, 0, 10, 20, display.height() - 1);
    display.drawFastHLine(0, y, display.width(), 0x001F);
  }

  int bar_width = display.width() / num_points - 4; // 2px pad to either side
  for (uint8_t i=0; i<num_points; i++) {
    int16_t x = map(i, 0, num_points, 0, display.width()) + 2; // Left edge of bar
    int16_t height = map(data[i], 0, 100, 0, display.height() - 20);
    // Some GFX functions (rects, H/V lines and similar) can accept negative
    // width/height values. What this does is anchor the shape at the right or
    // bottom coordinate (rather than the usual left/top) and draw back from
    // there, hence the -height here (bar is anchored at bottom of screen):
    display.fillRect(x, display.height() - 1, bar_width, -height, 0xFFE0);
  }

  delay(PAUSE);

} // END CHART EXAMPLES
```

```
// TEXT ALIGN FUNCTIONS ----------------------------------------------------

// Adafruit_GFX only handles left-aligned text. This is normal and by design;
// it's a rare need that would further strain AVR by incurring a ton of extra
// code to properly handle, and some details would confuse. If needed, these
// functions give a fair approximation, with the "gotchas" that multi-line
// input won't work, and this operates only as a println(), not print()
// (though, unlike println(), cursor X does not reset to column 0, instead
// returning to initial column and downward by font's line spacing). If you
// can work with those constraints, it's a modest amount of code to copy
// into a project. Or, if your project only needs one or two aligned strings,
// simply use getTextBounds() for a bounding box and work from there.
// DO NOT ATTEMPT TO MAKE THIS A GFX-NATIVE FEATURE, EVERYTHING WILL BREAK.

typedef enum { // Alignment options passed to functions below
  GFX_ALIGN_LEFT,
  GFX_ALIGN_CENTER,
  GFX_ALIGN_RIGHT
} GFXalign;

// Draw text aligned relative to current cursor position. Arguments:
// gfx   : An Adafruit_GFX-derived type (e.g. display or canvas object).
// str   : String to print (as a char *).
// align : One of the GFXalign values declared above.
//         GFX_ALIGN_LEFT is normal left-aligned println() behavior.
//         GFX_ALIGN_CENTER prints centered on cursor pos.
//         GFX_ALIGN_RIGHT prints right-aligned to cursor pos.
// Cursor advances down one line a la println(). Column is unchanged.
void print_aligned(Adafruit_GFX &gfx, const char *str,
                   GFXalign align = GFX_ALIGN_LEFT) {
  uint16_t w, h;
  int16_t  x, y, cursor_x, cursor_x_save;
  cursor_x = cursor_x_save = gfx.getCursorX();
  gfx.getTextBounds(str, 0, gfx.getCursorY(), &x, &y, &w, &h);
  if (align == GFX_ALIGN_RIGHT)       cursor_x -= w;
  else if (align == GFX_ALIGN_CENTER) cursor_x -= w / 2;
  //gfx.drawRect(cursor_x, y, w, h, 0xF800);      // Debug rect
  gfx.setCursor(cursor_x - x, gfx.getCursorY());  // Center/right align
  gfx.println(str);
  gfx.setCursor(cursor_x_save, gfx.getCursorY()); // Restore cursor X
}

// Equivalent function for strings in flash memory (e.g. F("Foo")). Body
// appears identical to above function, but with C++ overloading it it works
// from flash instead of RAM. Any changes should be made in both places.
void print_aligned(Adafruit_GFX &gfx, const __FlashStringHelper *str,
                   GFXalign align = GFX_ALIGN_LEFT) {
  uint16_t w, h;
  int16_t  x, y, cursor_x, cursor_x_save;
  cursor_x = cursor_x_save = gfx.getCursorX();
  gfx.getTextBounds(str, 0, gfx.getCursorY(), &x, &y, &w, &h);
  if (align == GFX_ALIGN_RIGHT)       cursor_x -= w;
  else if (align == GFX_ALIGN_CENTER) cursor_x -= w / 2;
  //gfx.drawRect(cursor_x, y, w, h, 0xF800);      // Debug rect
  gfx.setCursor(cursor_x - x, gfx.getCursorY());  // Center/right align
  gfx.println(str);
  gfx.setCursor(cursor_x_save, gfx.getCursorY()); // Restore cursor X
}

// Equivalent function for Arduino Strings; converts to C string (char *)
// and calls corresponding print_aligned() implementation.
void print_aligned(Adafruit_GFX &gfx, const String &str,
                   GFXalign align = GFX_ALIGN_LEFT) {
  print_aligned(gfx, const_cast<char *>(str.c_str()));
}

// TEXT EXAMPLES -----------------------------------------------------------

// This section demonstrates:
```

```
// - Using the default 5x7 built-in font, including scaling in each axis.
// - How to access all characters of this font, including symbols.
// - Using a custom font, including alignment techniques that aren't a normal
//   part of the GFX library (uses functions above).

void show_basic_text() {
  // Show text scaling with built-in font.
  display.fillScreen(0);
  display.setFont();                    // Use default font
  display.setCursor(0, CORNER_RADIUS); // Initial cursor position
  display.setTextSize(1);              // Default size
  display.println(F("Standard built-in font"));
  display.setTextSize(2);
  display.println(F("BIG TEXT"));
  display.setTextSize(3);
  // "BIGGER TEXT" won't fit on narrow screens, so abbreviate there.
  display.println((display.width() >= 200) ? F("BIGGER TEXT") : F("BIGGER"));
  display.setTextSize(2, 4);
  display.println(F("TALL and"));
  display.setTextSize(4, 2);
  display.println(F("WIDE"));

  delay(PAUSE);
} // END BASIC TEXT EXAMPLE

void show_char_map() {
  // "Code Page 437" is a name given to the original IBM PC character set.
  // Despite age and limited language support, still seen in small embedded
  // settings as it has some useful symbols and accented characters. The
  // default 5x7 pixel font of Adafruit_GFX is modeled after CP437. This
  // function draws a table of all the characters & explains some issues.

  // There are 256 characters in all. Draw table as 16 rows of 16 columns,
  // plus hexadecimal row & column labels. How big can each cell be drawn?
  const int cell_size = min(display.width(), display.height()) / 17;
  if (cell_size < 8) return; // Screen is too small for table, skip example.
  const int total_size = cell_size * 17; // 16 cells + 1 row or column label

  // Set up for default 5x7 font at 1:1 scale. Custom fonts are NOT used
  // here as most are only 128 characters to save space (the "7b" at the
  // end of many GFX font names means "7 bits," i.e. 128 characters).
  display.setFont();
  display.setTextSize(1);

  // Early Adafruit_GFX was missing one symbol, throwing off some indices!
  // But fixing the library would break MANY existing sketches that relied
  // on the degrees symbol and others. The default behavior is thus "broken"
  // to keep older code working. New code can access the CORRECT full CP437
  // table by calling this function like so:
  display.cp437(true);

  display.fillScreen(0);

  const int16_t x = (display.width() - total_size) / 2;  // Upper left corner of
  int16_t       y = (display.height() - total_size) / 2; // table centered on screen
  if (y >= 4) { // If there's a little extra space above & below, scoot table
    y += 4;     // down a few pixels and show a message centered at top.
    display.setCursor((display.width() - 114) / 2, 0); // 114 = pixel width
    display.print(F("CP437 Character Map"));           //   of this message
  }

  const int16_t inset_x = (cell_size - 5) / 2; // To center each character within
cell,
  const int16_t inset_y = (cell_size - 8) / 2; // compute X & Y offset from corner.

  for (uint8_t row=0; row<16; row++) { // 16 down...
    // Draw row and column headings as hexadecimal single digits. To get the
    // hex value for a specific character, combine the left & top labels,
    // e.g. Pi symbol is row E, column 3, thus: display.print((char)0xE3);
    display.setCursor(x + (row + 1) * cell_size + inset_x, y + inset_y);
    display.print(row, HEX); // This actually draws column labels
    display.setCursor(x + inset_x, y + (row + 1) * cell_size + inset_y);
    display.print(row, HEX); // and THIS is the row labels
    for (uint8_t col=0; col<16; col++) { // 16 across...
      if ((row + col) & 1) { // Fill alternating cells w/gray
        display.fillRect(x + (col + 1) * cell_size, y + (row + 1) * cell_size,
                         cell_size, cell_size, 0x630C);
      }
      // drawChar() bypasses usual cursor positioning to go direct to an X/Y
      // location. If foreground & background match, it's drawn transparent.
      display.drawChar(x + (col + 1) * cell_size + inset_x,
                       y + (row + 1) * cell_size + inset_y, row * 16 + col,
                       0xFFFF, 0xFFFF, 1);
    }
  }

  delay(PAUSE * 2);
} // END CHAR MAP EXAMPLE

void show_custom_text() {
  // Show use of custom fonts, plus how to do center or right alignment
  // using some additional functions provided earlier.

  display.fillScreen(0);
  display.setFont(&FreeSansBold18pt7b);
  display.setTextSize(1);
  display.setTextWrap(false); // Allow text off edges

  // Get "M height" of custom font and move initial base line there:
  uint16_t w, h;
  int16_t  x, y;
  display.getTextBounds("M", 0, 0, &x, &y, &w, &h);
  // On rounded 240x280 display in tall orientation, "Custom Font" gets
  // clipped by top corners. Scoot text down a few pixels in that one case.
  if (CORNER_RADIUS && (display.height() == 280)) h += 20;
  display.setCursor(display.width() / 2, h);

  if (display.width() >= 200) {
    print_aligned(display, F("Custom Font"), GFX_ALIGN_CENTER);
    display.setCursor(0, display.getCursorY() + 10);
    print_aligned(display, F("Align Left"), GFX_ALIGN_LEFT);
    display.setCursor(display.width() / 2, display.getCursorY());
    print_aligned(display, F("Centered"), GFX_ALIGN_CENTER);
    // Small rounded screen, when oriented the wide way, "Right" gets
    // clipped by bottom right corner. Scoot left to compensate.
    int16_t x_offset = (CORNER_RADIUS && (display.height() < 200)) ? 15 : 0;
    display.setCursor(display.width() - x_offset, display.getCursorY());
    print_aligned(display, F("Align Right"), GFX_ALIGN_RIGHT);
  } else {
    // On narrow screens, use abbreviated messages
    print_aligned(display, F("Font &"), GFX_ALIGN_CENTER);
    print_aligned(display, F("Align"), GFX_ALIGN_CENTER);
    display.setCursor(0, display.getCursorY() + 10);
    print_aligned(display, F("Left"), GFX_ALIGN_LEFT);
    display.setCursor(display.width() / 2, display.getCursorY());
    print_aligned(display, F("Center"), GFX_ALIGN_CENTER);
    display.setCursor(display.width(), display.getCursorY());
    print_aligned(display, F("Right"), GFX_ALIGN_RIGHT);
  }

  delay(PAUSE);
} // END CUSTOM FONT EXAMPLE

// BITMAP EXAMPLE ----------------------------------------------------------

// This section demonstrates:
// - Embedding a small bitmap in the code (flash memory).
// - Drawing that bitmap in various colors, and transparently (only '1' bits
//   are drawn; '0' bits are skipped, leaving screen contents in place).
// - Use of the color565() function to decimate 24-bit RGB to 16 bits.

#define HEX_WIDTH  16 // Bitmap width in pixels
#define HEX_HEIGHT 16 // Bitmap height in pixels
// Bitmap data. PROGMEM ensures it's in flash memory (not RAM). And while
// it would be valid to leave the brackets empty here (i.e. hex_bitmap[]),
// having dimensions with a little math makes the compiler verify the
// correct number of bytes are present in the list.
PROGMEM const uint8_t hex_bitmap[(HEX_WIDTH + 7) / 8 * HEX_HEIGHT] = {
  0b00000001, 0b10000000,
  0b00000111, 0b11100000,
  0b00011111, 0b11111000,
  0b01111111, 0b11111110,
  0b01111111, 0b11111110,
  0b01111111, 0b11111110,
  0b01111111, 0b11111110,
  0b01111111, 0b11111110,
  0b01111111, 0b11111110,
  0b01111111, 0b11111110,
  0b01111111, 0b11111110,
  0b01111111, 0b11111110,
  0b01111111, 0b11111110,
  0b00011111, 0b11111000,
  0b00000111, 0b11100000,
  0b00000001, 0b10000000,
};
#define Y_SPACING (HEX_HEIGHT - 2) // Used by code below for positioning

void show_bitmap() {
  display.fillScreen(0);

  // Not screen center, but UL coordinates of center hexagon bitmap
  const int16_t center_x = (display.width() - HEX_WIDTH) / 2;
  const int16_t center_y = (display.height() - HEX_HEIGHT) / 2;
  const uint8_t steps = min((display.height() - HEX_HEIGHT) / Y_SPACING,
                            display.width() / HEX_WIDTH - 1) / 2;

  display.drawBitmap(center_x, center_y, hex_bitmap, HEX_WIDTH, HEX_HEIGHT,
                     0xFFFF); // Draw center hexagon in white

  // Tile the hexagon bitmap repeatedly in a range of hues. Don't mind the
  // bit of repetition in the math, the optimizer easily picks this up.
  // Also, if math looks odd, keep in mind "PEMDAS" operator precedence;
  // multiplication and division occur before addition and subtraction.
  for (uint8_t a=0; a<=steps; a++) {
    for (uint8_t b=1; b<=steps; b++) {
      display.drawBitmap( // Right section centered red: a = green, b = blue
        center_x + (a + b) * HEX_WIDTH / 2,
        center_y + (a - b) * Y_SPACING,
        hex_bitmap, HEX_WIDTH, HEX_HEIGHT,
        display.color565(255, 255 - 255 * a / steps, 255 - 255 * b / steps));
      display.drawBitmap( // UL section centered green: a = blue, b = red
        center_x - b * HEX_WIDTH + a * HEX_WIDTH / 2,
        center_y - a * Y_SPACING,
        hex_bitmap, HEX_WIDTH, HEX_HEIGHT,
        display.color565(255 - 255 * b / steps, 255, 255 - 255 * a / steps));
      display.drawBitmap( // LL section centered blue: a = red, b = green
        center_x - a * HEX_WIDTH + b * HEX_WIDTH / 2,
        center_y + b * Y_SPACING,
        hex_bitmap, HEX_WIDTH, HEX_HEIGHT,
        display.color565(255 - 255 * a / steps, 255 - 255 * b / steps, 255));
    }
  }

  delay(PAUSE);
} // END BITMAP EXAMPLE

// CANVAS EXAMPLE ----------------------------------------------------------
// This section demonstrates:
// - How to refresh changing values onscreen without erase/redraw flicker.
// - Using an offscreen canvas. It's similar to a bitmap above, but rather
//   than a fixed pattern in flash memory, it's drawable like the screen.
// - More tips on text alignment, and adapting to different screen sizes.

#define PADDING 6 // Pixels between axis label and value

void show_canvas() {
  // For this example, let's suppose we want to display live readings from a
  // sensor such as a three-axis accelerometer, something like:
  //   X: (number)
  //   Y: (number)
  //   Z: (number)
  // To look extra classy, we want a custom font, and the labels for each
  // axis are right-aligned so the ':' characters line up...

  display.setFont(&FreeSansBold18pt7b); // Use a custom font
  display.setTextSize(1);               // and reset to 1:1 scale

  const char *label[] = { "X:", "Y:", "Z:" };       // Labels for each axis
  const uint16_t color[] = { 0xF800, 0x07E0, 0x001F }; // Colors for each value

  // To get the labels right-aligned, one option would be simple trial and
  // error to find a column that looks good and doesn't clip anything off.
  // Let's do this dynamically though, so it adapts to any font or labels!
  // Start by finding the widest of the label strings:
  uint16_t w, h, max_w = 0;
  int16_t  x, y;
  for (uint8_t i=0; i<3; i++) { // For each label...
    display.getTextBounds(label[i], 0, 0, &x, &y, &w, &h);
    if (w > max_w) max_w = w; // Keep track of widest label
  }

  // Rounded corners throwing us a curve again. If needed, scoot everything
  // to the right a bit on wide displays, down a bit on tall ones.
  int16_t y_offset = 0;
  if (display.width() > display.height()) max_w += CORNER_RADIUS;
  else                                    y_offset = CORNER_RADIUS;

  // Now we have max_w for right-aligning the labels. Before we draw them
  // though...in order to perform flicker-free updates, the numbers we show
  // will be rendered in either a GFXcanvas1 or GFXcanvas16 object; a 1-bit
  // or 16-bit offscreen bitmap, RAM permitting. The correct size for this
  // canvas could also be trial-and-errored, but again let's make this adapt
  // automatically. The width of the canvas will span from max_w (plus a few
  // pixels for padding) to the right edge. But the height? Looking at an
  // uppercase 'M' can work in many situations, but some fonts have ascenders
  // and descenders on digits, and in some locales a comma (extending below
  // the baseline) is the decimal separator. Feed ALL the numeric chars into
  // getTextBounds() for a cumulative height:
  display.setTextWrap(false); // Keep on one line
  display.getTextBounds(F("0123456789.,-"), 0, 0, &x, &y, &w, &h);

  // Now declare a GFXcanvas16 object based on the computed width & height:
  GFXcanvas16 canvas16(display.width() - max_w - PADDING, h);

  // Small devices (e.g. ATmega328p) will almost certainly lack enough RAM
  // for the canvas. Check if canvas buffer exists. If not, fall back on
  // using a 1-bit (rather than 16-bit) canvas. Much more RAM friendly, but
  // not as fast to draw. If a project doesn't require super interactive
  // updates, consider just going straight for the more compact Canvas1.
  if (canvas16.getBuffer()) {
    // If here, 16-bit canvas allocated successfully! Point of interest,
    // only one canvas is needed for this example, we can reuse it for all
    // three numbers because the regions are the same size.

    // display and canvas are independent drawable objects; must explicitly
    // set the same custom font to use on the canvas now:
    canvas16.setFont(&FreeSansBold18pt7b);

    // Clear display and print labels. Once drawn, these remain untouched.
    display.fillScreen(0);
    display.setCursor(max_w, -y + y_offset); // Set baseline for first row
    for (uint8_t i=0; i<3; i++) print_aligned(display, label[i], GFX_ALIGN_RIGHT);

    // Last part now is to print numbers on the canvas and copy the canvas to
    // the display, repeating for several seconds...
    uint32_t elapsed, startTime = millis();
    while ((elapsed = (millis() - startTime)) <= PAUSE * 2) {
      for (uint8_t i=0; i<3; i++) {  // For each label...
        canvas16.fillScreen(0);    // fillScreen() in this case clears canvas
        canvas16.setCursor(0, -y); // Reset baseline for custom font
        canvas16.setTextColor(color[i]);
        // These aren't real accelerometer readings, just cool-looking numbers.
        // Notice we print to the canvas, NOT the display:
        canvas16.print(sin(elapsed / 200.0 + (float)i * M_PI * 2.0 / 3.0), 5);
        // And HERE is the secret sauce to flicker-free updates. Canvas details
        // can be passed to the drawRGBBitmap() function, which fully overwrites
        // prior screen contents in that area. yAdvance is font line spacing.
        display.drawRGBBitmap(max_w + PADDING, i * FreeSansBold18pt7b.yAdvance +
                              y_offset, canvas16.getBuffer(), canvas16.width(),
                              canvas16.height());
      }
    }
  } else {
    // Insufficient RAM for Canvas16. Try declaring a 1-bit canvas instead...
    GFXcanvas1 canvas1(display.width() - max_w - PADDING, h);
    // If even this smaller object fails, can't proceed, cancel this example.
    if (!canvas1.getBuffer()) return;

    // Remainder here is nearly identical to the code above, simply using a
    // different canvas type. It's stripped of most comments for brevity.
    canvas1.setFont(&FreeSansBold18pt7b);
    display.fillScreen(0);
    display.setCursor(max_w, -y + y_offset);
    for (uint8_t i=0; i<3; i++) print_aligned(display, label[i], GFX_ALIGN_RIGHT);
    uint32_t elapsed, startTime = millis();
    while ((elapsed = (millis() - startTime)) <= PAUSE * 2) {
      for (uint8_t i=0; i<3; i++) {
        canvas1.fillScreen(0);
        canvas1.setCursor(0, -y);
        canvas1.print(sin(elapsed / 200.0 + (float)i * M_PI * 2.0 / 3.0), 5);
        // Here's the secret sauce to flicker-free updates with GFXcanvas1.
        // Canvas details can be passed to the drawBitmap() function, and by
        // specifying both a foreground AND BACKGROUND color (0), this will fully
        // overwrite/erase prior screen contents in that area (vs transparent).
        display.drawBitmap(max_w + PADDING, i * FreeSansBold18pt7b.yAdvance +
                           y_offset, canvas1.getBuffer(), canvas1.width(),
                           canvas1.height(), color[i], 0);
      }
    }
  }

  // Because canvas object was declared locally to this function, it's freed
  // automatically when the function returns; no explicit delete needed.
} // END CANVAS EXAMPLE
```

The example code uses the product ID number of your display to determine which library and attributes to select when running the code. This section is located at the top of the code. By default, the code uses product ID `5393`.

```
// *** EDIT THIS VALUE TO MATCH THE ADAFRUIT PRODUCT ID FOR YOUR DISPLAY: ***
#define SCREEN_PRODUCT_ID 5393
// You can find the product ID several ways:
// - "PID" accompanies each line-item on your receipt or order details page.
// - Visit adafruit.com and search for EYESPI displays. On product pages,
//   PID is shown just below product title, and is at the end of URLs.
// - Check the comments in setup() later that reference various screens.
```

To have the code work for the ILI9341 240x320 2.2" TFT, change `SCREEN_PRODUCT_ID` to `1480`.

```
// *** EDIT THIS VALUE TO MATCH THE ADAFRUIT PRODUCT ID FOR YOUR DISPLAY: ***
#define SCREEN_PRODUCT_ID 1480
// You can find the product ID several ways:
// - "PID" accompanies each line-item on your receipt or order details page.
// - Visit adafruit.com and search for EYESPI displays. On product pages,
//   PID is shown just below product title, and is at the end of URLs.
// - Check the comments in setup() later that reference various screens.
```

After updating the `SCREEN_PRODUCT_ID`, upload the sketch to your board. You should see the graphics test begin running on your display. The tests include drawing shapes, graphs and text. After every loop of the test, the code will rotate the screen orientation by 90 degrees and run the test again. The code is heavily commented so you can utilize the examples for various graphic techniques in your projects.

## Arduino Docs

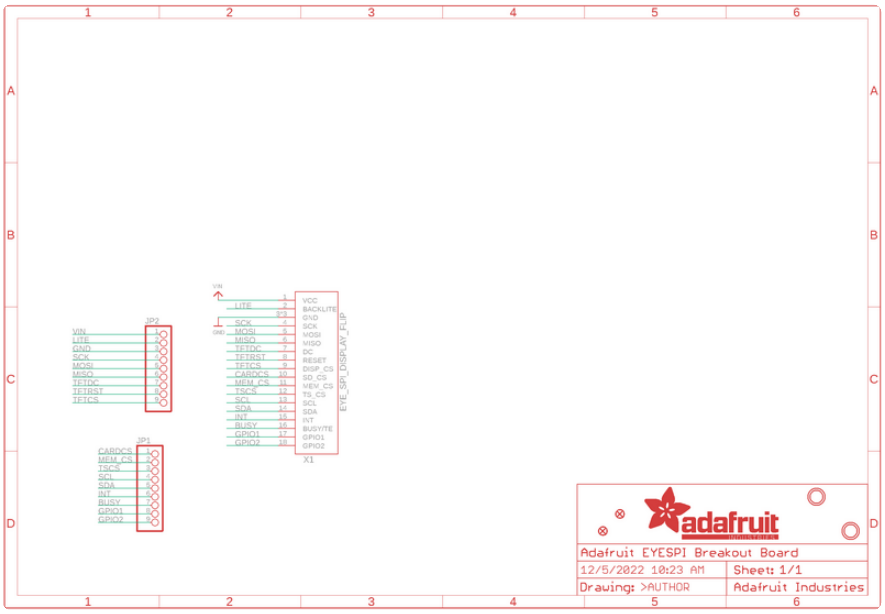
[Arduino Docs ()]

## Downloads

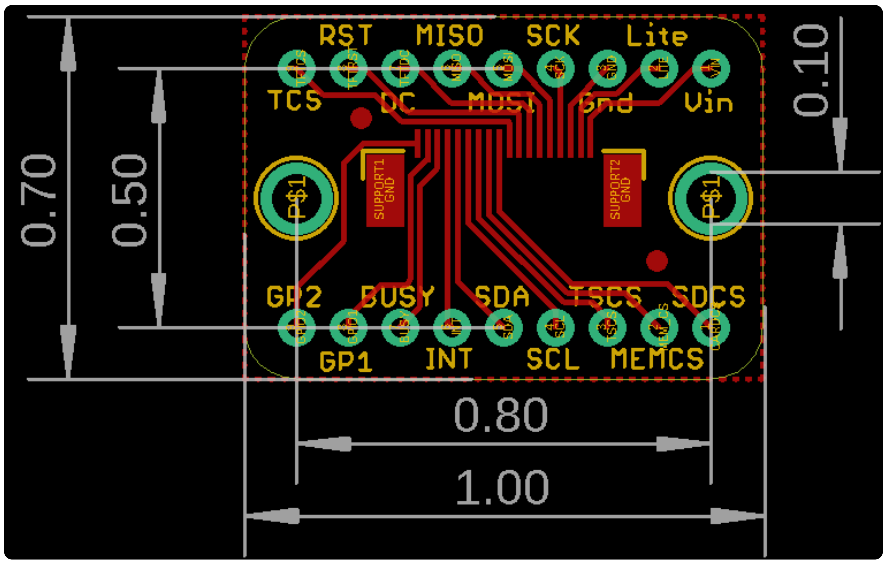
### Files

- EagleCAD PCB files on GitHub ()
- Fritzing object in the Adafruit Fritzing Library ()
- Fritzing object with EYESPI cable in the Adafruit Fritzing Library ()

### Schematic and Fab Print



adafruit
Adafruit EYESPI Breakout Board
12/5/2022 10:23 AM
Sheet: 1/1
Drawing: >AUTHOR
Adafruit Industries



RST
MISO
SCK
Lite
TCS
DC
MOSI
Gnd
Vin
GP2
BUSY
SDA
TSCS
SDCS
GP1
INT
SCL
MEMCS
0.70
0.50
0.10
0.80
1.00