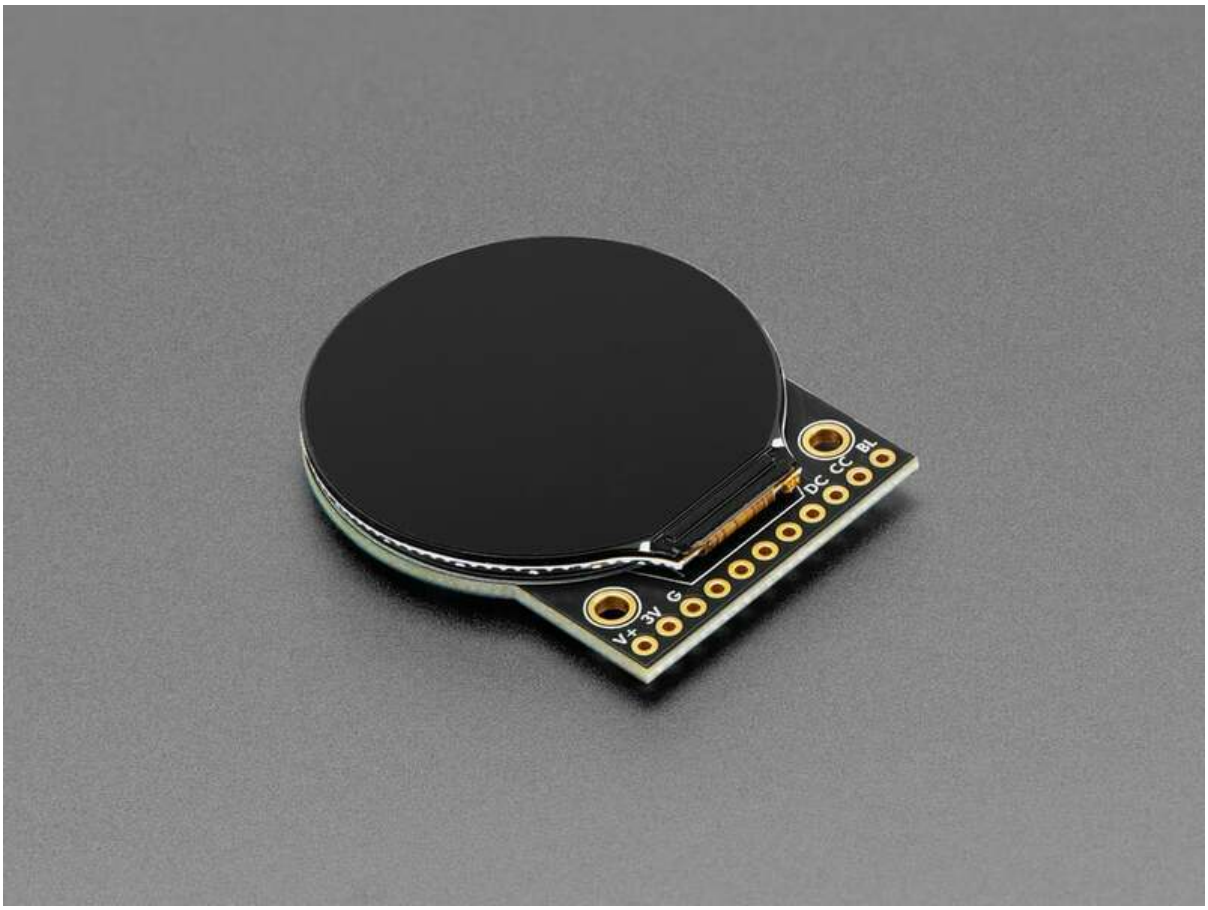
Adafruit 1.28" 240x240 Round TFT LCD

Created by Liz Clark

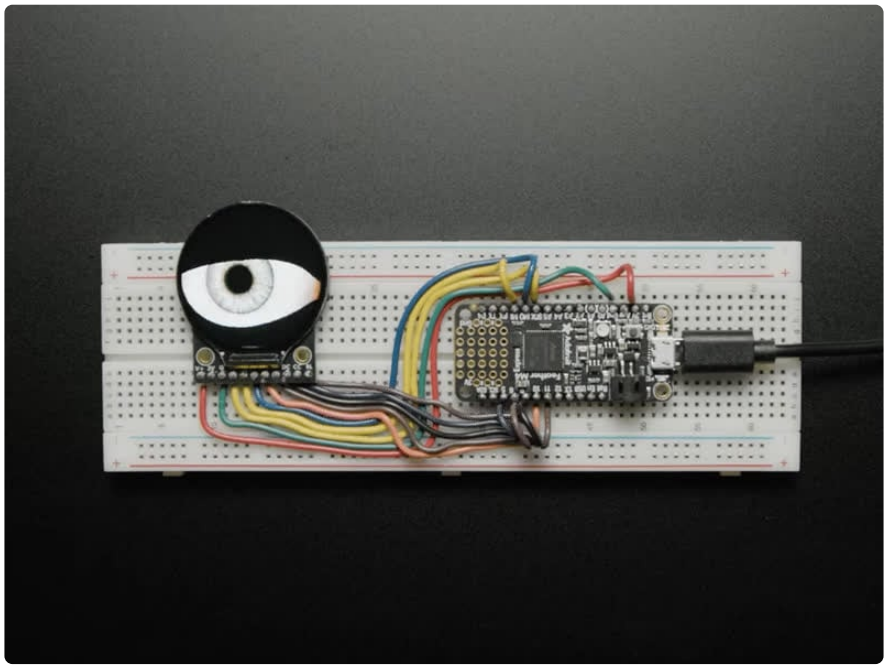


https://learn.adafruit.com/adafruit-1-28-240x240-round-tft-lcd

Last updated on 2025-04-14 03:59:16 PM EDT

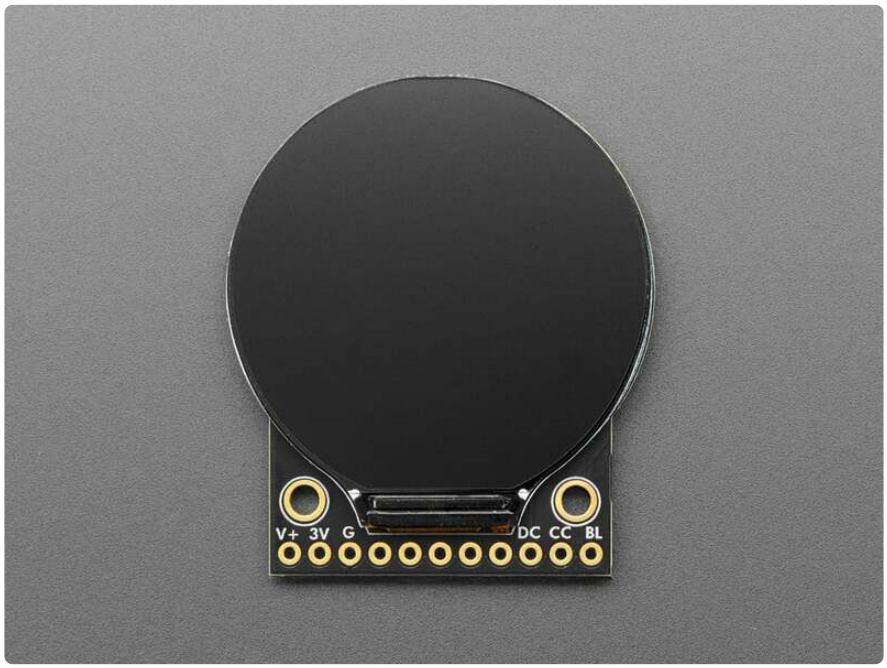
Table of Contents

Overview

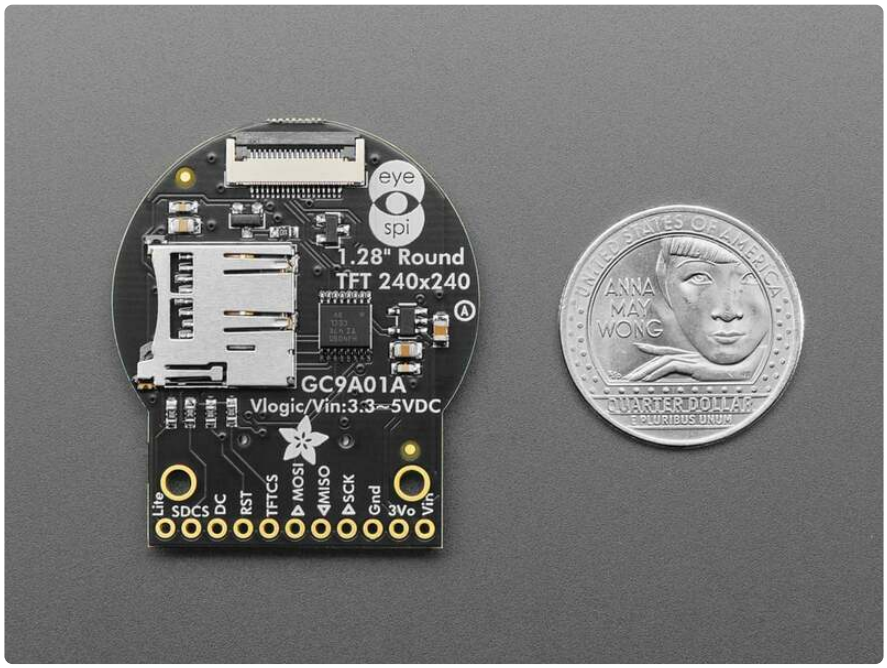


'Round these parts we enjoy unusually-shaped displays. And this one certainly fits the description - it's a 1.28" diagonal TFT that comes in a round shape and contains a high density 220 ppi, 240x240 pixel RGB display with full-angle viewing.

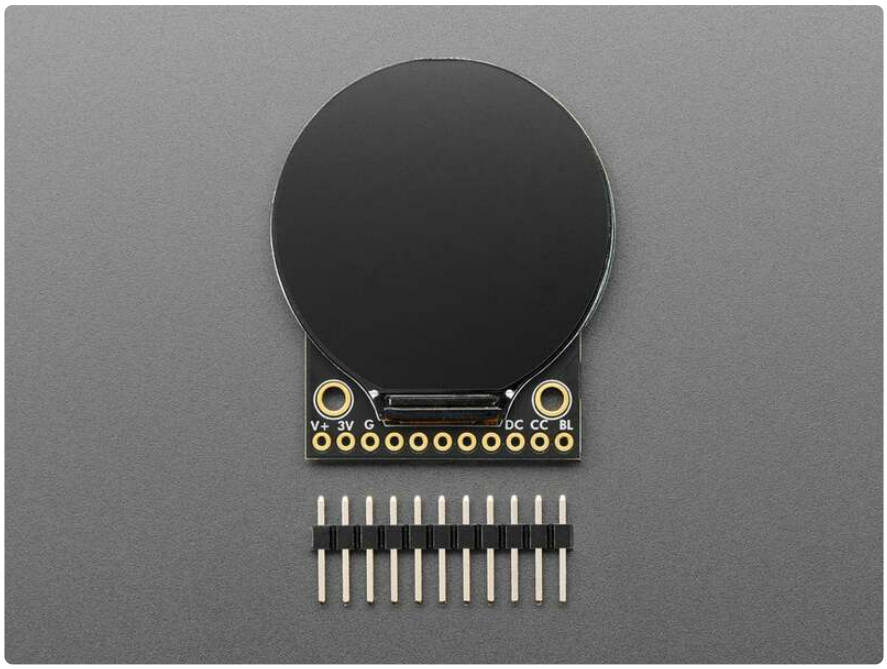
It looks a lot like our 1.54" 240x240 square display (http://adafru.it/3787), but with the edges beveled off. We've seen displays like this often used in smartwatches and small electronic devices but they've always been MIPI interface. Finally, we found one that is SPI and has a friendly display driver, so it works with any and all microcontrollers or microcomputers!



This lovely little display breakout is the best way to add a small, round, colorful and very bright display to any project. Since the display uses 4-wire SPI to communicate and has its own pixel-addressable frame buffer, it can be used with every kind of microcontroller. Even a very small one with low memory and few pins available! The 1.28" display has 240x240 16-bit full color pixels and is an **IPS** display, so the color looks great up to 80 degrees off axis in any direction. The TFT driver (GC9A01A) is very similar to the popular ST7789, [and our Arduino library supports it well](https://adafru.it/1aem) (https://adafru.it/1aem).



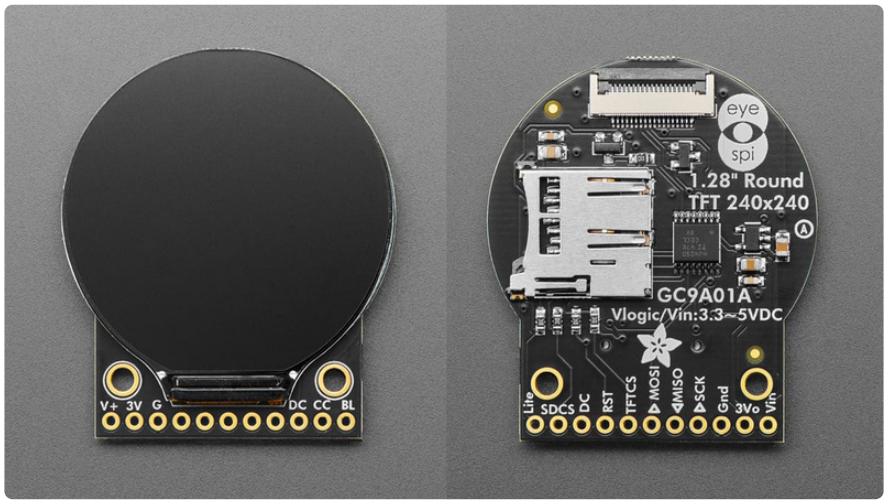
The breakout has the TFT display soldered on (it uses a delicate flex-circuit connector) as well as an ultra-low-dropout 3.3V regulator, auto-reset circuitry, and a 3/5V level shifter so you can use it with 3.3V or 5V power and logic. We also had a little extra space, so we placed a microSD card holder so you can easily load full color bitmaps from a FAT16/FAT32 formatted microSD card. The microSD card is not included, [but you can pick one up here](http://adafru.it/5251) (http://adafru.it/5251).



Of course, we wouldn't just leave you with a datasheet and a "good luck!" - we've written a full open-source graphics Arduino library that can draw pixels, lines, rectangles, circles, text, and bitmaps as well as example code (https://adafru.it/1aem). The code is written for Arduino but can be easily ported to your favorite microcontroller! Wiring is easy, we strongly encourage using the hardware SPI pins of your Arduino as software SPI is noticeably slower when dealing with this size display. For Raspberry Pi or other Single Board Computer Python users, we have a user-space Pillow-compatible library (https://adafru.it/u1C). For CircuitPython there's a contributed displayio driver for native support (https://adafru.it/1aen).

This display breakout also features a 18-pin "EYESPI" standard FPC connector (https://adafru.it/18fg) with flip-top connector. You can use a 18-pin 0.5mm pitch FPC cable (http://adafru.it/5239) **(not included!)** to connect to all the GPIO pins, for when you want to skip the soldering.

Pinouts



Power Pins

- **V+ / Vin** - This is the power pin. To power the board, give it the same power as the logic level of your microcontroller - e.g. for a 3V microcontroller like a Feather M4, use 3V, or for a 5V microcontroller like Arduino, use 5V.
- **3V / 3Vo** - This is the output from the onboard 3.3V regulator. If you have a need for a clean 3.3V output, you can use this! It can provide at least 100mA output.
- **G / Gnd** - This is common ground for power and logic.

SPI Pins

- **SCK** - this is the **S**PI **C**lock pin. Use 3-5V logic level.
- **MOSI** - this is the **S**erial **Da**ta In / **M**icrocontroller **O**ut **S**ensor **I**n pin. It is used to send data from the microcontroller to the SD card and/or TFT. Use either a 3.3 or 5V logic level
- **MISO** - this is the **S**erial **D**ata **O**ut / **M**icrocontroller **I**n **S**ensor **O**ut pin. It is used for the SD card. It isn't used for the TFT display which is write-only. It is 3.3V logic out (but can be read by 5V logic)
- **TFTCS** - this is the **TFT C**hip **S**elect pin. Use either a 3.3 or 5V logic level

Other Pins

- **RST** - This is the TFT reset pin. Connect to ground to reset the TFT! 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 Reset pin, which works for most cases. There is an automatic-reset chip connected so it will reset on power-up. Use either a 3.3 or 5V logic level
- **DC** - This is the TFT SPI data or command selector pin. Use either a 3.3 or 5V logic level

- **CC / SDCS** - This is the SD card chip select pin, used if you want to read from the SD card. Use either a 3.3 or 5V logic level
- **BL / 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. Use either a 3.3V or 5V logic level

micro SD Card Slot

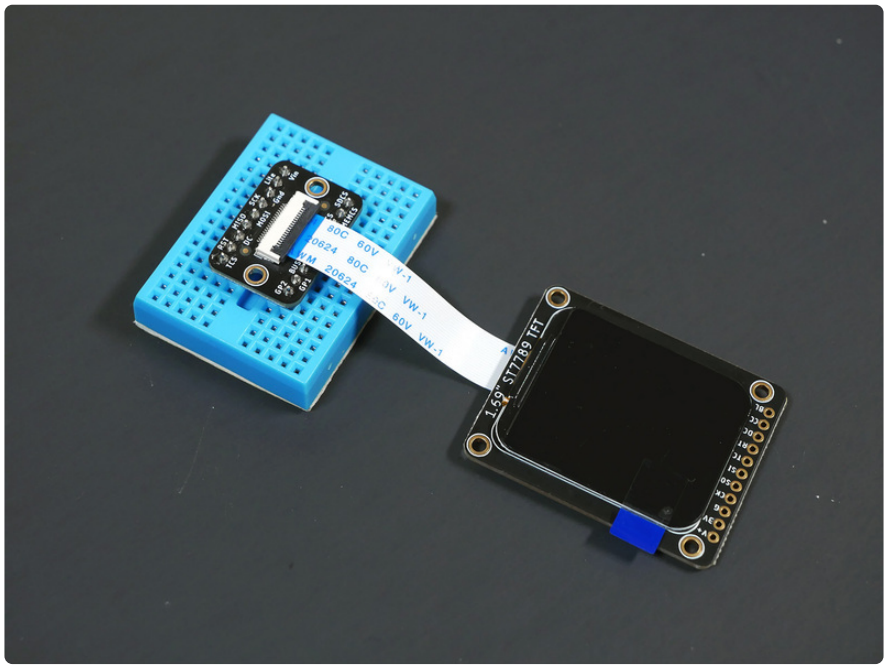
- On the back of the board, towards the middle, is the micro SD card slot. You can use any micro SD card that supports SPI mode with one CS pin.

EYESPI Connector

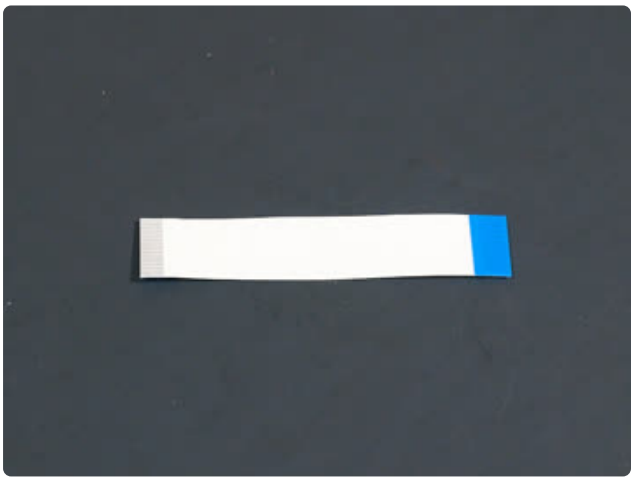
At the top edge of the back of the display is the EYESPI connector. This 18-pin FPC connector allows you to connect EYESPI-compatible displays using an EYESPI cable, with no soldering or jumper wires needed. All of the pins broken out at the bottom of the display board are available through the EYESPI connector:

1. VIN (3 to 5V DC power)
2. Backlight (Use either a 3.3 or 5V logic level, PWM optional input)
3. Ground
4. SPI Clock (Use either a 3.3 or 5V logic level in)
5. SPI MOSI (Use either a 3.3 or 5V logic level Microcontroller Out, Screen/SD In)
6. SPI MISO (Use either a 3.3 or 5V logic level Microcontroller In, Screen/SD Out)
7. TFT Data/Command (Use either a 3.3 or 5V logic level in)
8. TFT Reset (optional either a 3.3 or 5V logic level in)
9. TFT SPI Chip Select (Use either a 3.3 or 5V logic level in)
10. SD Card SPI Chip Select (Use either a 3.3 or 5V logic level in)
11. Unused
12. Unused
13. Unused
14. Unused
15. Unused
16. Unused
17. Unused
18. Unused

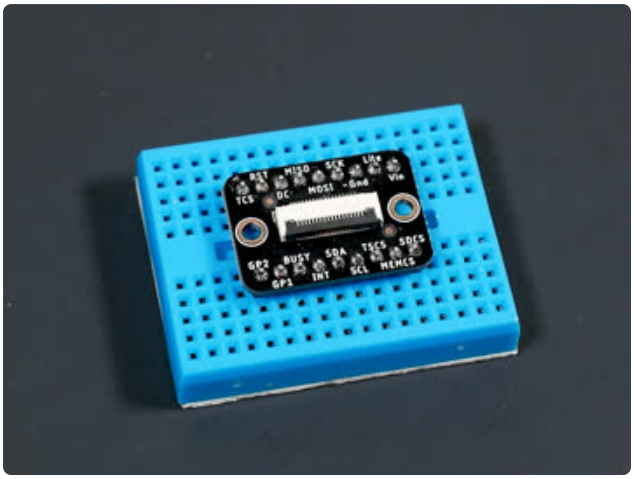
Plugging in an EYESPI Cable



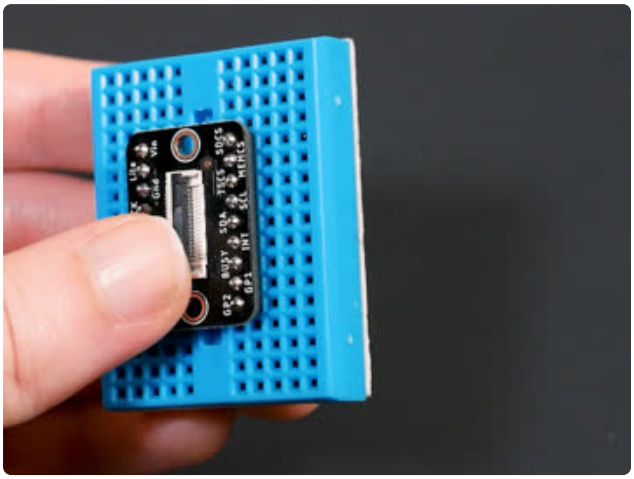
You can connect an EYESPI compatible display to the EYESPI breakout board using an EYESPI cable. An EYESPI cable is an 18 pin flexible PCB (FPC). The FPC can only be connected properly in one orientation, so be sure to follow the steps below to ensure that your display and breakout are plugged in properly.



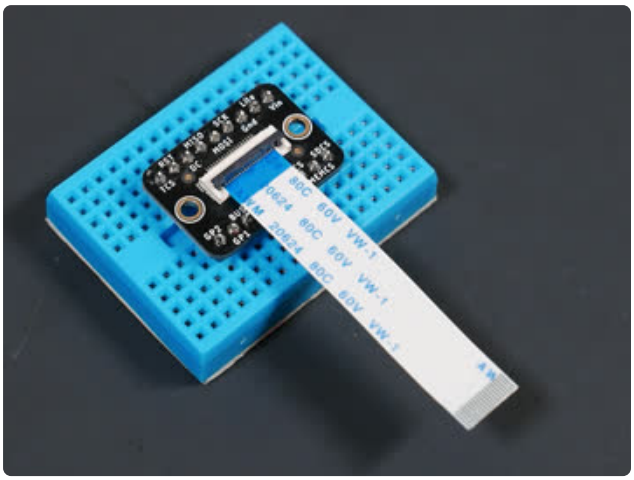
Each EYESPI cable has **blue stripes** on either end. On the other side of the cable, underneath the blue stripe, are the connector pins that make contact with the FPC connector pins on the display or breakout.



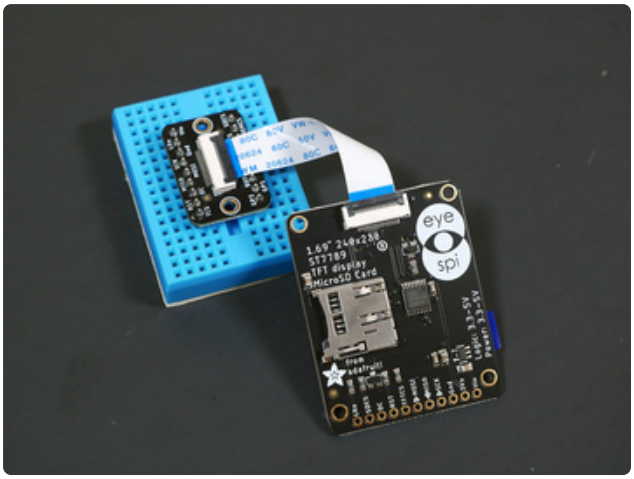
To begin inserting an EYESPI cable to an FPC connector, gently lift the FPC connector black latch up.



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.

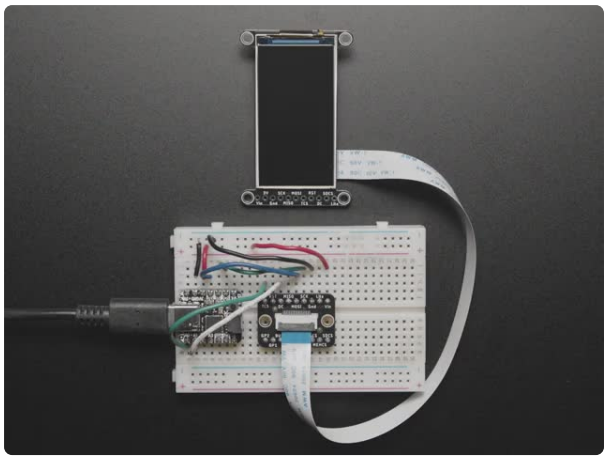


Repeat this process for the FPC connector on your display. Again, ensure that the **blue stripe** on either end of the cable is facing up.

CircuitPython

Using the 240x240 round TFT with CircuitPython involves wiring up the display to your board and loading the code and necessary libraries onto your board to see the graphics test on the display.

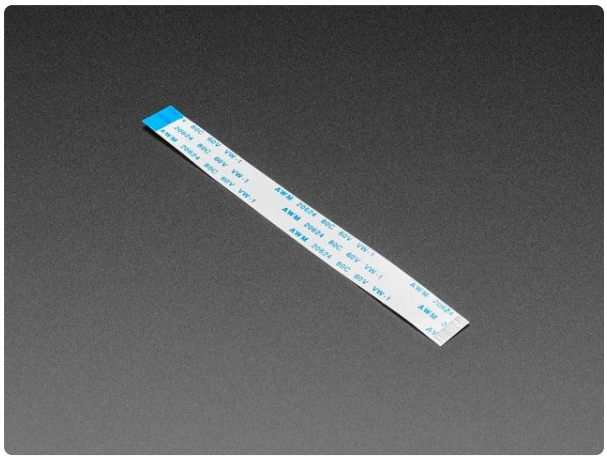
You can wire up the display with a breadboard or using an EYESPI breakout board.



Adafruit EYESPI Breakout Board - 18 Pin FPC Connector

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

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



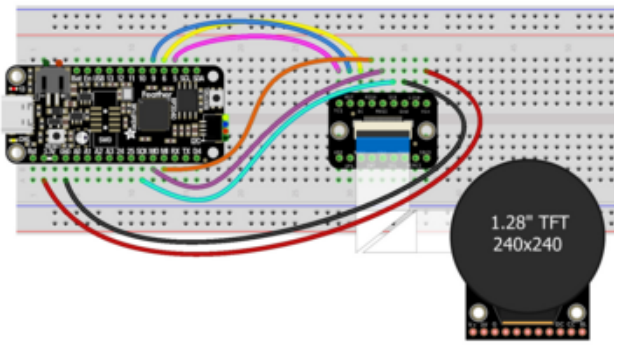
EYESPI Cable - 18 Pin 100mm long Flex PCB (FPC) A-B type

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

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

Wiring

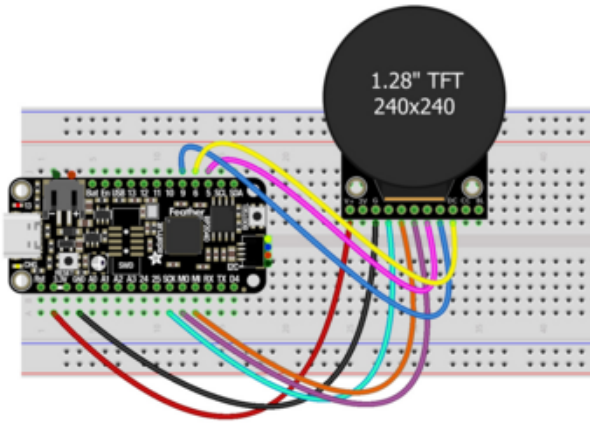
First, wire the TFT to your CircuitPython-compatible microcontroller. The diagram below shows wiring up to a Feather RP2040 with an EYESPI breakout:



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

Attach the **TFT screen to the EYESPI breakout** with an **EYESPI cable** as described on the Plugging in an EYESPI Cable (https://adafru.it/1aeo) page.

The following is the TFT wired to a Feather RP2040 using a solderless breadboard:



Feather 3.3V to **TFT V+ (red wire)**
Feather GND to **TFT G (black wire)**
Feather SCK to **TFT SCK (cyan wire)**
Feather MO to **TFT MOSI (purple wire)**
Feather MI to **TFT MISO (orange wire)**
Feather D9 to **TFT RST (blue wire)**
Feather D6 to **TFT DC (yellow wire)**
Feather D5 to **TFT TCS (pink wire)**

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.

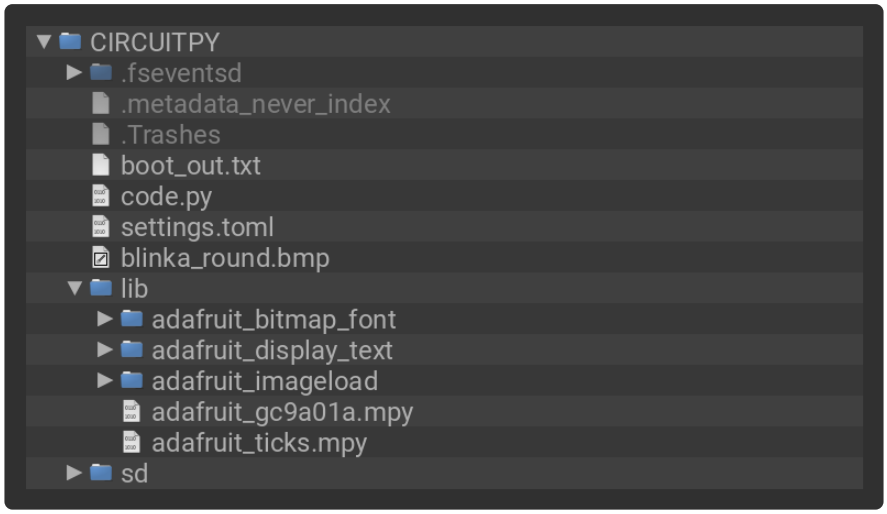
Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file.

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**, **blinka_round.bmp** bitmap 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_imageload**
- **adafruit_gc9a01a.mpy**
- **adafruit_ticks.mpy**

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



Example Code

```
# SPDX-FileCopyrightText: 2025 Liz Clark for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time
import board
import displayio
import terminalio
from adafruit_display_text.bitmap_label import Label
import adafruit_imageload
from fourwire import FourWire
from vectorio import Circle
from adafruit_gc9a01a import GC9A01A

spi = board.SPI()
tft_cs = board.D5
tft_dc = board.D6
tft_reset = board.D9

displayio.release_displays()

display_bus = FourWire(spi, command=tft_dc, chip_select=tft_cs, reset=tft_reset)
```

```
display = GC9A01A(display_bus, width=240, height=240)

# --- Default Shapes/Text Demo ---
main_group = displayio.Group()
display.root_group = main_group

bg_bitmap = displayio.Bitmap(240, 240, 2)
color_palette = displayio.Palette(2)
color_palette[0] = 0x00FF00  # Bright Green
color_palette[1] = 0xAA0088  # Purple

bg_sprite = displayio.TileGrid(bg_bitmap, pixel_shader=color_palette, x=0, y=0)
main_group.append(bg_sprite)

inner_circle = Circle(pixel_shader=color_palette, x=120, y=120, radius=100,
color_index=1)
main_group.append(inner_circle)

text_group = displayio.Group(scale=2, x=50, y=120)
text = "Hello World!"
text_area = Label(terminalio.FONT, text=text, color=0xFFFF00)
text_group.append(text_area)  # Subgroup for text scaling
main_group.append(text_group)

# --- ImageLoad Demo ---
blinka_group = displayio.Group()
bitmap, palette = adafruit_imageload.load("/blinka_round.bmp",
                                          bitmap=displayio.Bitmap,
                                          palette=displayio.Palette)

grid = displayio.TileGrid(bitmap, pixel_shader=palette)
blinka_group.append(grid)

while True:
    # show shapes/text
    display.root_group = main_group
    time.sleep(2)
    # show blinka bitmap
    display.root_group = blinka_group
    time.sleep(2)
```

After running the code, you should see a green circle appear on your display, followed by a smaller, inset purple circle, and finally, yellow text centered on the display saying, "Hello World!". This is followed by a bitmap of Blinka in her ouroboros circular pose. The two graphics will alternate being displayed every 2 seconds.

CircuitPython Docs

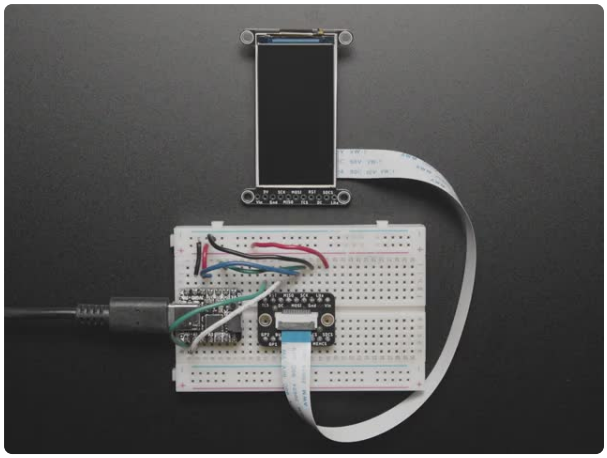
CircuitPython Docs (https://adafru.it/1aek)

Python

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.

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).

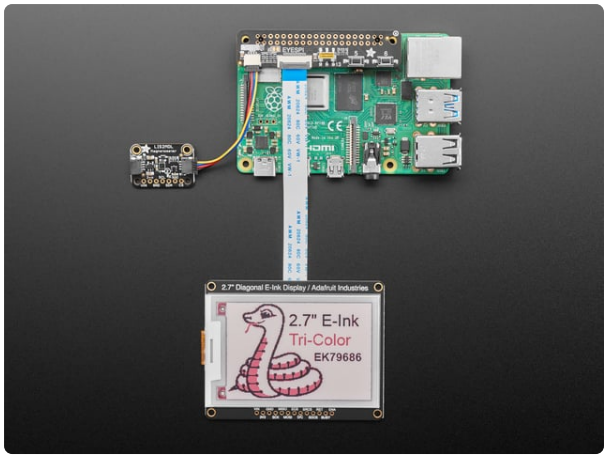
You can wire up the TFT to your Raspberry Pi using a breadboard, EYESPI breakout (http://adafru.it/5613) or EYESPI Pi Beret (http://adafru.it/5783).



Adafruit EYESPI Breakout Board - 18 Pin FPC Connector

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

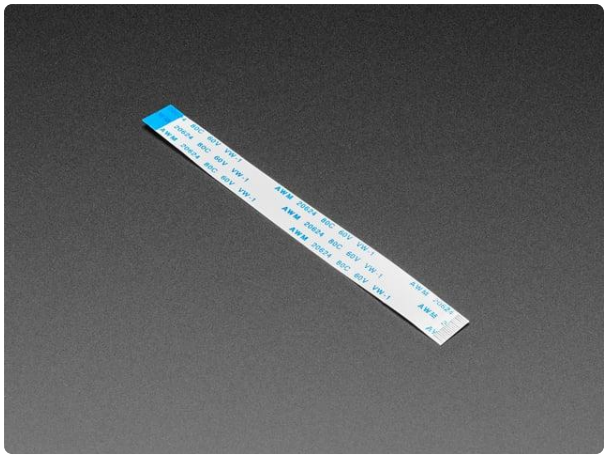
https://www.adafruit.com/product/5613



[Adafruit EYESPI Pi Beret - Buttons, EYESPI and STEMMA QT]

Raspberry Pi's make for handy lil computers, but they're really wonderful when you can connect all sorts of nifty hardware to them: color TFT or E-Ink displays, and sensors are...

https://www.adafruit.com/product/5783



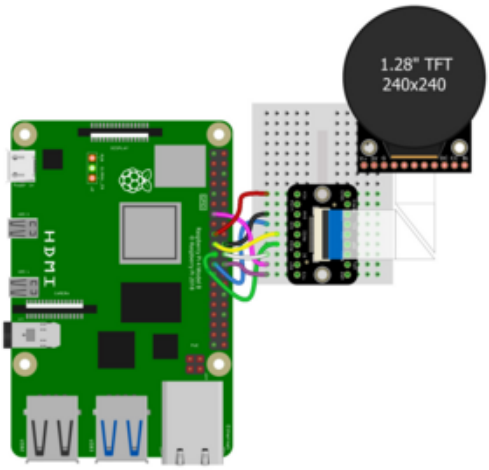
[EYESPI Cable - 18 Pin 100mm long Flex PCB (FPC) A-B type]

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

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

Wiring

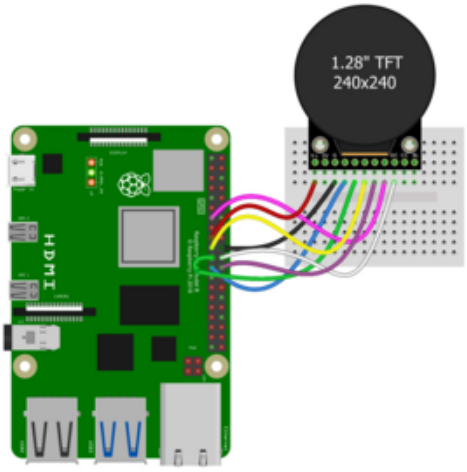
Here is how you'll connect your Raspberry Pi to the round TFT with an EYESPI breakout.



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 27 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 (https://adafru.it/1aeo) page.

The following is the TFT wired to a Raspberry Pi using a solderless breadboard:



Pi 3V to **TFT V+ (red wire)**
Pi GND to **TFT G (black wire)**
Pi SLCK (GPIO 11) to **TFT SCK (blue wire)**
Pi MISO (GPIO 9) to **TFT MISO (green wire)**
Pi MOSI (GPIO 10) to **TFT MOSI (yellow wire)**
Pi CE0 (GPIO 8) to **TFT TCS (purple wire)**
Pi GPIO 25 to **TFT DC (white wire)**
Pi GPIO 27 to **TFT RST (pink wire)**

Software 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)!

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

Run the following command to install the DejaVu font:

- `sudo apt-get install fonts-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`

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

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

Blinka JPEG

Save the JPEG image of Blinka in the same directory as your **code.py** file. You can download the image from here or through the Project Bundle from GitHub.

Python Usage

Once you have all of the requirements installed on your Raspberry Pi, copy or download the following example, and run the following, replacing **code.py** with whatever you named the file:

```
python3 code.py
```

Example Code

```
# SPDX-FileCopyrightText: 2025 Liz Clark for Adafruit Industries
# SPDX-FileCopyrightText: Adapted from Melissa LeBlanc-Williams's Pi Demo Code
#
# SPDX-License-Identifier: MIT

'''Raspberry Pi Graphics example for the 240x240 Round Display'''

import time
import digitalio
import board
from PIL import Image, ImageDraw, ImageFont
from adafruit_rgb_display import gc9a01a

BORDER = 20
FONTSIZE = 24

cs_pin = digitalio.DigitalInOut(board.CE0)
dc_pin = digitalio.DigitalInOut(board.D25)
reset_pin = digitalio.DigitalInOut(board.D27)

BAUDRATE = 24000000


spi = board.SPI()

disp = gc9a01a.GC9A01A(spi, rotation=0,
    width=240, height=240,
    x_offset=0, y_offset=0,
    cs=cs_pin,
    dc=dc_pin,
    rst=reset_pin,
    baudrate=BAUDRATE,
)

width = disp.width
height = disp.height
```

```
# -------TEXT AND SHAPES---------
image1 = Image.new("RGB", (width, height))
draw1 = ImageDraw.Draw(image1)
draw1.ellipse((0, 0, width, height), fill=(0, 255, 0))  # Green background

draw1.ellipse(
    (BORDER, BORDER, width - BORDER - 1, height - BORDER - 1), fill=(170, 0, 136)
)
font = ImageFont.truetype("/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf",
FONTSIZE)
text = "Hello World!"
(font_width, font_height) = font.getsize(text)
draw1.text(
    (width // 2 - font_width // 2, height // 2 - font_height // 2),
    text,
    font=font,
    fill=(255, 255, 0),
)

# ------ADABOT JPEG DISPLAY----------
image2 = Image.open("blinka_round.jpg")
image_ratio = image2.width / image2.height
screen_ratio = width / height
scaled_width = width
scaled_height = image2.height * width // image2.width
image2 = image2.resize((scaled_width, scaled_height), Image.BICUBIC)
x = scaled_width // 2 - width // 2
y = scaled_height // 2 - height // 2
image2 = image2.crop((x, y, x + width, y + height))

while True:
    disp.image(image1)  # show text
    time.sleep(2)
    disp.image(image2)  # show adabot
    time.sleep(2)
```

You'll see the graphics example show on the TFT. First, you'll see a purple circle on top of a green circle with the text "Hello World!" in the center. Then, you'll see the Blinka JPEG displayed. These two images will swap back and forth in the loop.

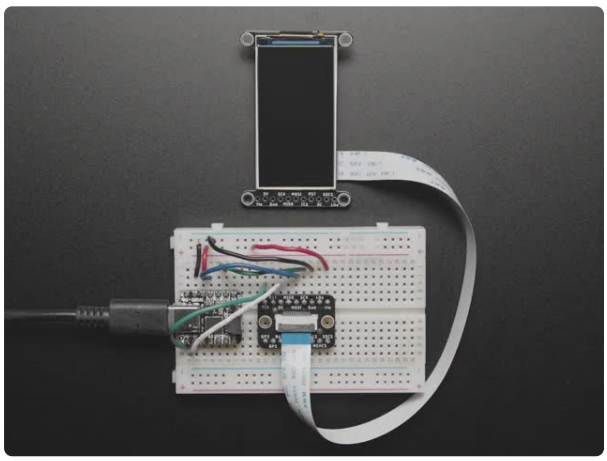
Python Docs

Python Docs (https://adafru.it/18kb)

Arduino

Using the 240x240 Round TFT Display with Arduino involves wiring up the display to your Arduino-compatible board, installing the libraries and running the provided example code.

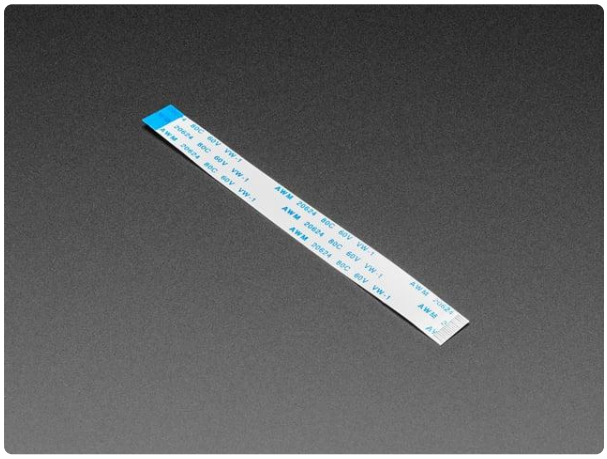
You can wire up the display with a breadboard or using an EYESPI breakout board.



Adafruit EYESPI Breakout Board - 18 Pin FPC Connector

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

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



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

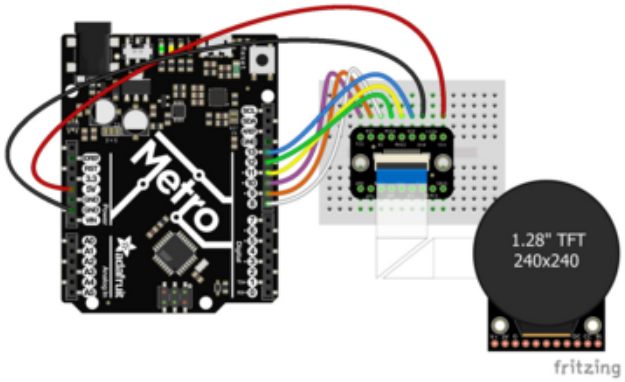
Connect this to that when a 18-pin FPC connector is needed. This 25 cm long cable is made of a flexible PCB. It's A-B style which means that pin one on one side will match...

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

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 breakout VDD.

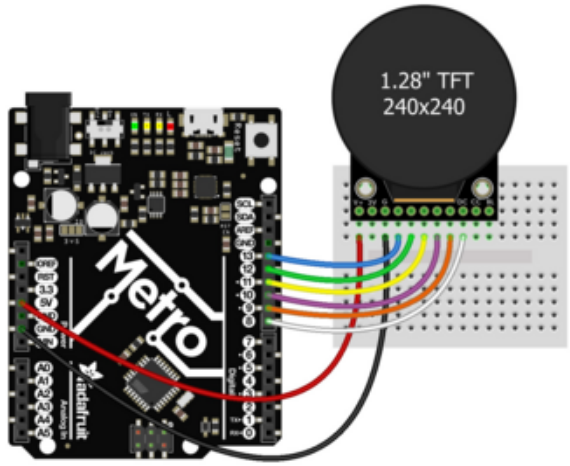
Here is an Adafruit Metro wired up to the TFT with an EYESPI breakout.



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 (https://adafru.it/1aeo) page.

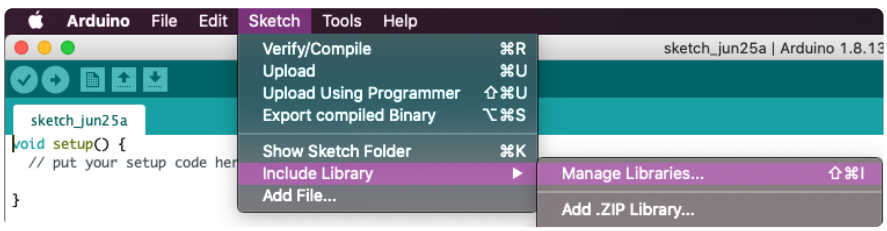
The following is the TFT wired to an Adafruit Metro using a solderless breadboard:



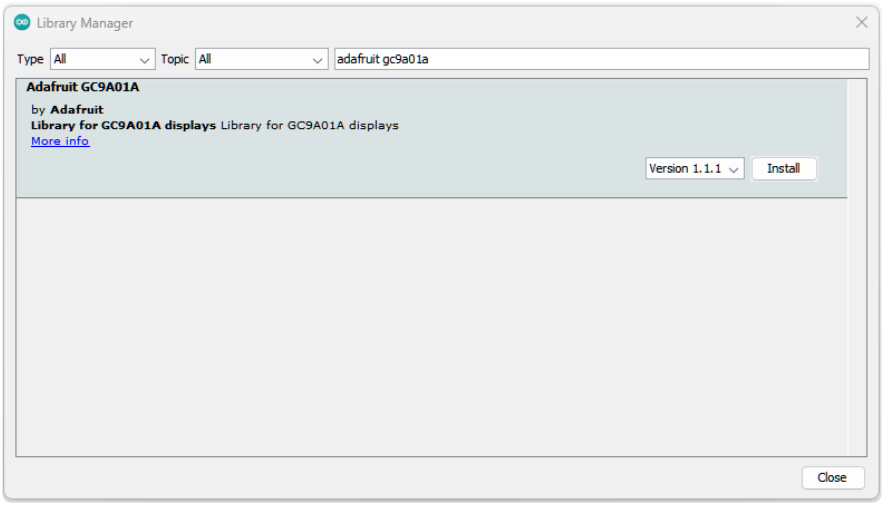
Board 5V to **TFT VIN (red wire)**
Board GND to **TFT GND (black wire)**
Board pin 13 to **TFT SCK (blue wire)**
Board pin 12 to **TFT MISO (green wire)**
Board pin 11 to **TFT MOSI (yellow wire)**
Board pin 10 to **TFT TCS (purple wire)**
Board pin 9 to **TFT RST (orange wire)**
Board pin 8 to **TFT DC (white wire)**

Library Installation

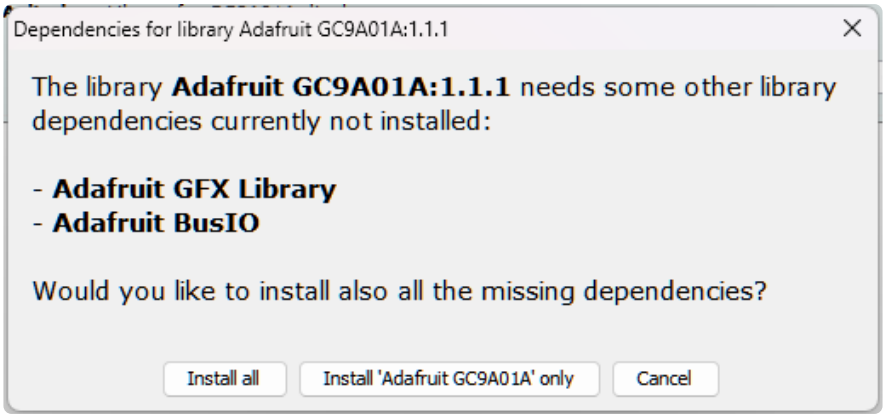
You can install the **Adafruit GC9A01A** library for Arduino using the Library Manager in the Arduino IDE.



Click the **Manage Libraries ...** menu item, search for **Adafruit GC9A01A**, and select the **Adafruit GC9A01A** library:



If asked about dependencies, click "Install all".



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 6178 // GC9A01A 240x240 round display
// 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 <SPI.h>
#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);
#elif (SCREEN_PRODUCT_ID == 6178)
#include "Adafruit_GC9A01A.h"
Adafruit_GC9A01A 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) // 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);
#elif (SCREEN_PRODUCT_ID == 6113)
  display.init(135, 240);
#else
#define CORNER_RADIUS 45// All ILI9341 & GC9A01A TFTs (320x240)/(240x240)
  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() {
  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.
  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

  // Rectangle
  display.fillScreen(0);
  display.drawRect(cx - half, cy - half, size, size, 0xF800);
  delay(500);

  display.fillScreen(0);
  display.fillRect(cx - half, cy - half, size, size, 0xF800);
  delay(500);

  // Triangle
  display.fillScreen(0);
  display.drawTriangle(
    cx, cy - half,           // Top
    cx - half, cy + half,    // Bottom left
    cx + half, cy + half,    // Bottom right
    0x07E0
  );
  delay(500);

  display.fillScreen(0);
  display.fillTriangle(
    cx, cy - half,           // Top
    cx - half, cy + half,    // Bottom left
    cx + half, cy + half,    // Bottom right
    0x07E0
  );
  delay(500);
```

```
  // Circle
  display.fillScreen(0);
  display.drawCircle(cx, cy, half, 0x001F);
  delay(500);

  display.fillScreen(0);
  display.fillCircle(cx, cy, half, 0x001F);
  delay(500);

  // Rounded Rectangle
  display.fillScreen(0);
  display.drawRoundRect(cx - half, cy - half, size, size, size/5, 0xFFE0);
  delay(500);

  display.fillScreen(0);
  display.fillRoundRect(cx - half, cy - half, size, size, size/5, 0xFFE0);
  delay(500);
}

// 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);

} // 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(50, CORNER_RADIUS); // Initial cursor position
  display.setTextSize(1);               // Default size
  display.println(F("Standard built-in font"));
  display.setTextSize(2);
  display.setCursor(30, display.getCursorY());
  display.println(F("BIG TEXT"));
  display.setTextSize(3);
  // "BIGGER TEXT" won't fit on narrow screens, so abbreviate there.
  display.setCursor(20, display.getCursorY());
  display.println((display.width() >= 200) ? F("BIGGER TEXT") : F("BIGGER"));
  display.setTextSize(2, 4);
  display.setCursor(15, display.getCursorY());
  display.println(F("TALL and"));
  display.setCursor(15, display.getCursorY());
  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 columm 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)) || (SCREEN_PRODUCT_ID == 6178)) h
+= 20;
  display.setCursor(display.width() / 2, h);

  if ((display.width() >= 200) && (SCREEN_PRODUCT_ID != 6178)) {
    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 {
    display.setCursor(display.width() / 2, display.height() / 2);
    print_aligned(display, F("Custom Font"), GFX_ALIGN_CENTER);

  }

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

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

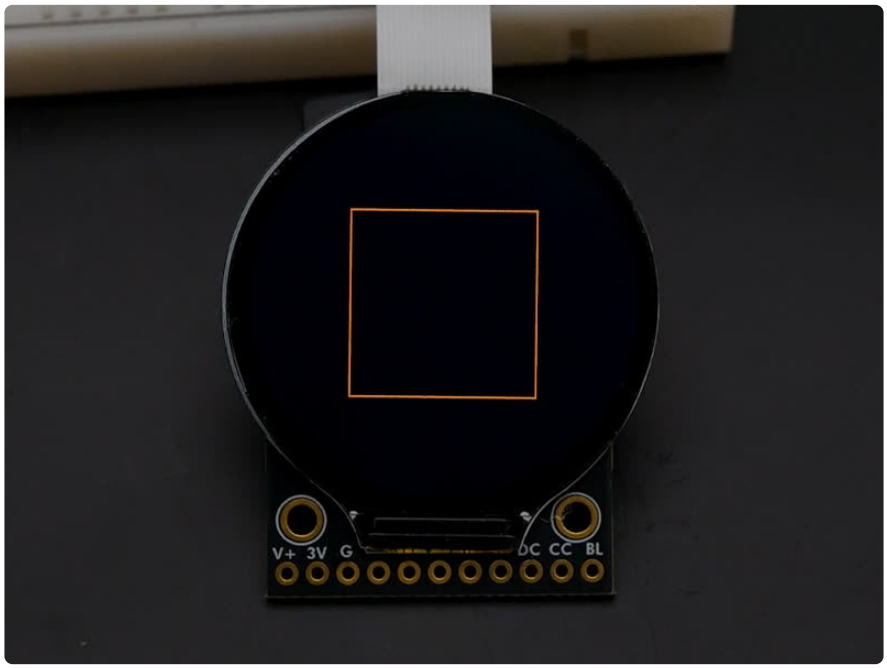
```
// This section demonstrates:
// - Embedding a small bitmap in the code (flash memory).
// - Drawing that bitmap in various colors, and transparently (only '1' bits
//   are drawn; '0' bits are skipped, leaving screen contents in place).
// - Use of the color565() function to decimate 24-bit RGB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

Upload the sketch to your board. You should see a modified 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 (https://adafru.it/1ael)

Downloads

Files

- GC9A01A Datasheet (https://adafru.it/1aep)
- TFT Display Datasheet (http://adafru.it/6178128901)
- EagleCAD PCB files on GitHub (https://adafru.it/1aeq)
- 3D models on GitHub (https://adafru.it/1ahl)
- Fritzing object in the Adafruit Fritzing Library (https://adafru.it/1aer)

Schematic and Fab Print