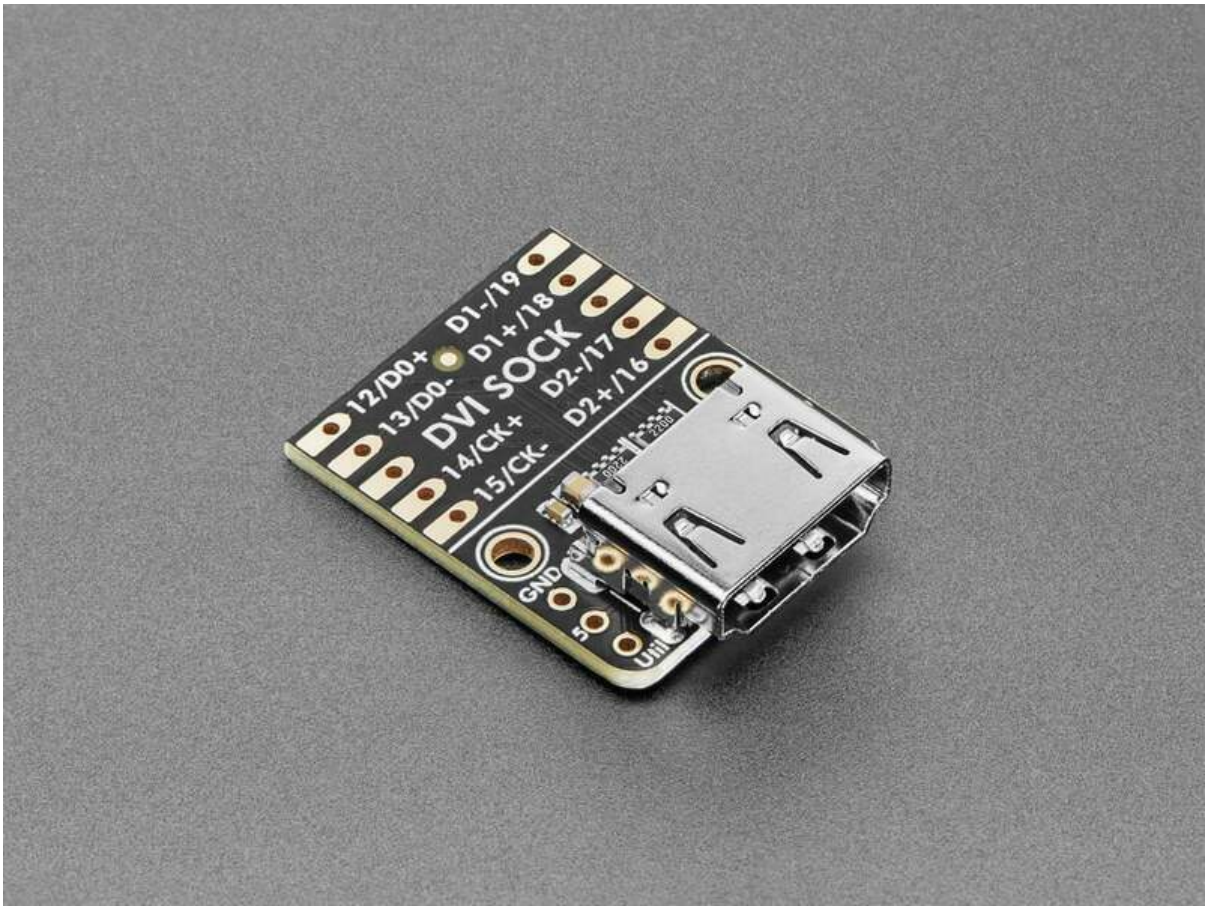
Adafruit DVI Sock for Pico

Created by Liz Clark

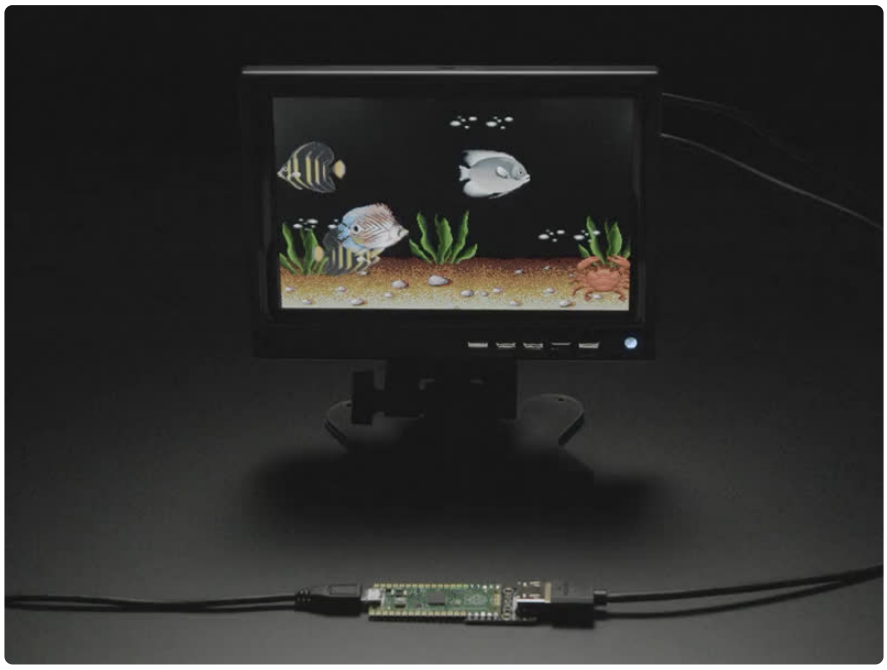


https://learn.adafruit.com/adafruit-dvi-sock-for-pico

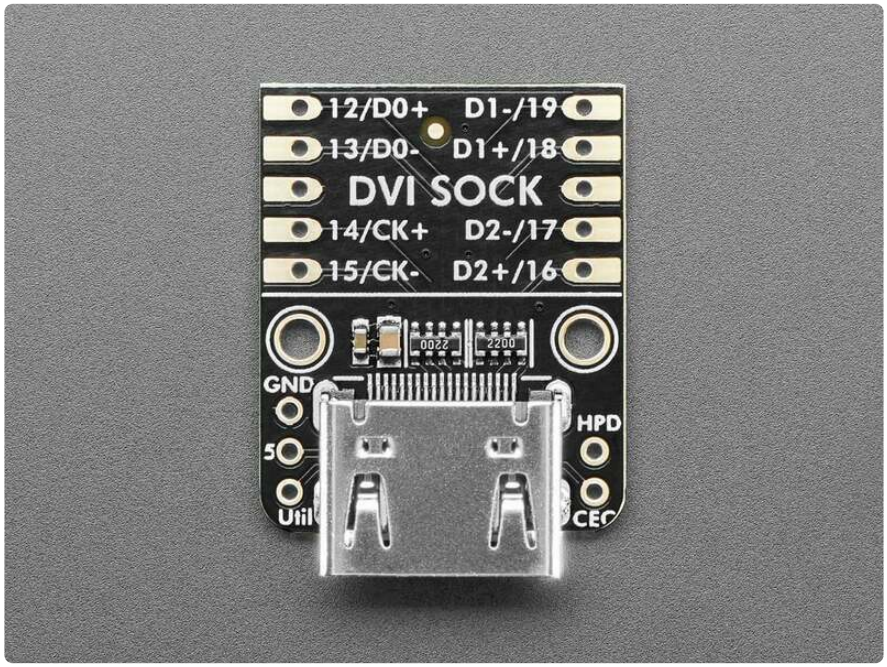
Last updated on 2024-06-03 04:00:35 PM EDT

Table of Contents

Overview

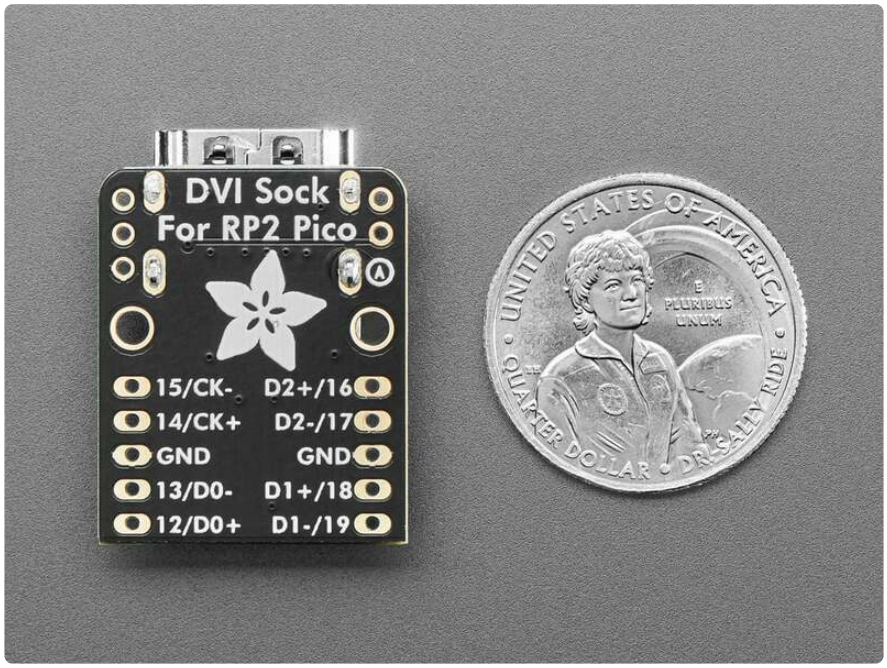


Wouldn't it be cool to display images and graphics directly from your Pico or Pico W to an HDMI monitor or television? We think so! So we designed this **DVI Sock** with a digital video output (a.k.a DVI) that will work with any HDMI monitor or display. Note it doesn't do audio, just graphics!



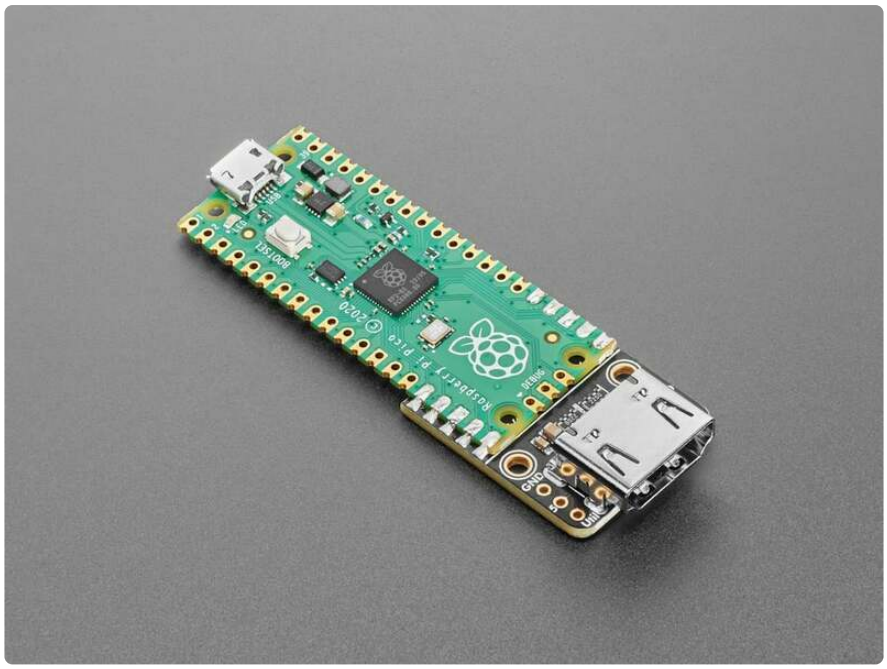
We designed this little breakout board after seeing Wren6991's Pico DVI Sock (https://adafru.it/19Wb) and their cool demos (https://adafru.it/ZTf)of the Raspberry Pi Pico driving an HDMI display. By using some fun 'abuse' of overclocking and the RP2040's

PIO system, a low-cost microcontroller can have great-looking video output. It's great for making demos or just noodling around with digital video generation.



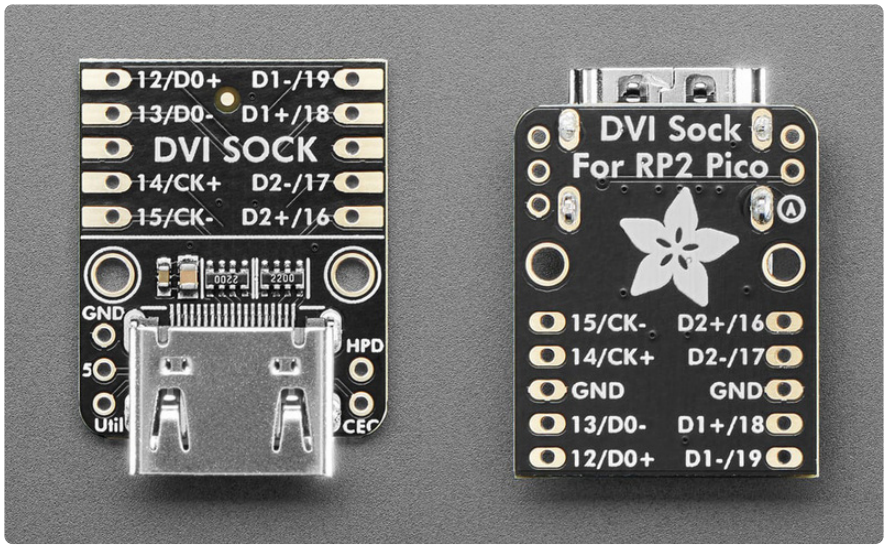
This breakout board has no active components on it. It's just a connector you can plug an HDMI/DVI cable into, and 220 ohm series resistors. Wire it up to the bottom pins of your Pico board, or solder it directly onto the 'end' like a li'l PCB sock. The connections that are made:

- GP12 to D0+
- GP13 to D0-
- GP14 to CK+
- GP15 to CK-
- GP16 to D2+
- GP17 to D2-
- GP18 to D1+
- GP19 to D1-
- Pico GND to GND



We also breakout the 5V, GND, Hot-Plug Detect, CEC and Util pins, for more advanced hacking. Note that while technically you're supposed to provide 5V to the DVI port, it isn't convenient to do so on the Sock so it's left disconnected. Since the 5V is used for the I2C EDID EEPROM and we don't care about it, the 'Sock will work just fine without.

Pinouts



Power Pins

- **5** - This is the 5V power pin. It is not connected by default. It is available for use if you're adding an I2C EEPROM to the board.
- **GND pins** - These are common ground for power and logic.

HDMI Connector

On the end of the board is the HDMI connector. It provides DVI output to any HDMI display or monitor. The following GPIO pins are routed to the connector:

- **GP12**: **D0+**
- **GP13**: **D0-**
- **GP14**: **CK+**
- **GP15**: **CK-**
- **GP16**: **D2+**
- **GP17**: **D2-**
- **GP18**: **D1+**
- **GP19**: **D1-**

Additional HDMI Pins

Three additional pins for the HDMI connector are broken out on either side of the HDMI port. You can route them to GPIO pins if you would like to use them:

- **Utility pin** - labeled **Util** on the board silk. This pin is reserved for future HDMI specification updates
- **CEC pin** - labeled **CEC** on the board silk. Consumer Electronic Control (https://adafru.it/18AQ) is a one-wire bidirectional serial bus that is standardized for remote control functions
- **Hot Plug Detection pin** - labeled **HPD** on the board silk. Hot plug detection is used to detect if a device is connected or disconnected to the HDMI connector by monitoring power, plug and unplug events

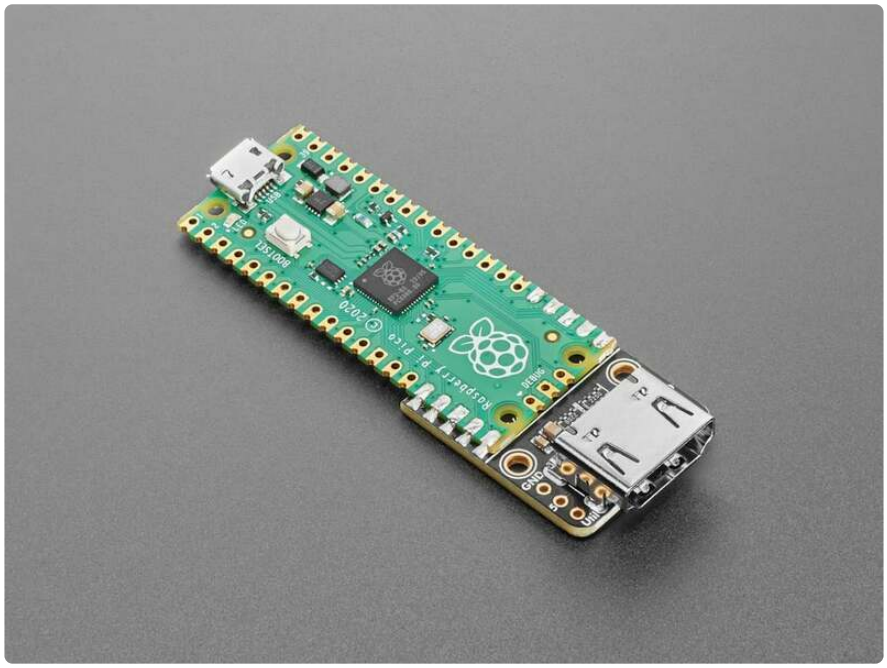
---

CircuitPython

It's easy to use the **DVI Sock** with CircuitPython and the PicoDVI (https://adafru.it/18Eu) core module. This module has been added to CircuitPython as of 8.1.0b2 (https://adafru.it/18Et) - but note that it uses a lot of memory so in particular if you want to use Pico W with WiFi support, you'll likely only be able to get away with monochrome display.

Wiring

Attach the Sock to the bottom pins of your Pico board, or solder it directly onto the 'end' like a li'l PCB sock.



CircuitPython Usage

To use with CircuitPython, you need to first install the PicoDVI dependencies into the **lib** folder onto your **CIRCUITPY** drive. Then you need to update **code.py** with the example script.

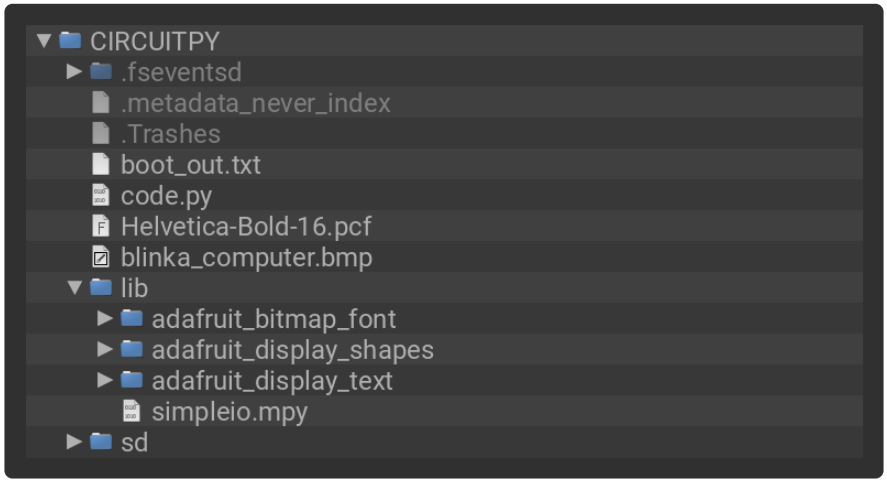
Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file.

Connect your Pico board to your computer via a known good data+power USB cable. The board should show up in your File Explorer/Finder (depending on your operating system) as a flash drive named **CIRCUITPY**.

Extract the contents of the zip file, and copy the **entire lib folder**, the **Helvetica-Bold-16.pcf** font file, **blinka_computer.bmp** bitmap file and **code.py** file to your **CIRCUITPY** drive.

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

- **adafruit_bitmap_font/**
- **adafruit_display_shapes/**
- **adafruit_display_text/**
- **simpleio.mpy**



Hello World DVI Output Example

Once everything is saved to the **CIRCUITPY** drive, you can connect the DVI Sock to an HDMI monitor and connect your Pico to USB power. You'll see the Hello World example display on the screen.

```
# SPDX-FileCopyrightText: 2023 Liz Clark for Adafruit Industries
# SPDX-FileCopyrightText: Adapted from Phil B.'s 16bit_hello Arduino Code
#
# SPDX-License-Identifier: MIT

import gc
import math
from random import randint
import time
import displayio
import picodvi
import board
import framebufferio
import vectorio
import terminalio
import simpleio
from adafruit_bitmap_font import bitmap_font
from adafruit_display_text import label, wrap_text_to_lines
from adafruit_display_shapes.rect import Rect
from adafruit_display_shapes.circle import Circle
from adafruit_display_shapes.roundrect import RoundRect
from adafruit_display_shapes.triangle import Triangle
from adafruit_display_shapes.line import Line

# pin defs for DVI Sock
displayio.release_displays()
fb = picodvi.Framebuffer(320, 240,
```

```
    clk_dp=board.GP14, clk_dn=board.GP15,
    red_dp=board.GP12, red_dn=board.GP13,
    green_dp=board.GP18, green_dn=board.GP19,
    blue_dp=board.GP16, blue_dn=board.GP17,
    color_depth=8)
display = framebufferio.FramebufferDisplay(fb)

bitmap = displayio.Bitmap(display.width, display.height, 3)

red = 0xff0000
yellow = 0xcccc00
orange = 0xff5500
blue = 0x0000ff
pink = 0xff00ff
purple = 0x5500ff
white = 0xffffff
green =  0x00ff00
aqua = 0x125690

palette = displayio.Palette(3)
palette[0] = 0x000000 # black
palette[1] = white
palette[2] = yellow

palette.make_transparent(0)

tile_grid = displayio.TileGrid(bitmap, pixel_shader=palette)

group = displayio.Group()

def clean_up(group_name):
    for _ in range(len(group_name)):
        group_name.pop()
    gc.collect()

def show_shapes():
    gc.collect()
    cx = int(display.width / 2)
    cy = int(display.height / 2)
    minor = min(cx, cy)
    pad = 5
    size = minor - pad
    half = int(size / 2)
    rect = Rect(cx - minor, cy - minor, size, size, stroke = 1, fill=red, outline =
red)
    tri = Triangle(cx + pad, cy - pad, cx + pad + half, cy - minor,
                   cx + minor - 1, cy - pad, fill=green, outline = green)
    circ = Circle(cx - pad - half, cy + pad + half, half, fill=blue, stroke = 1,
outline = blue)
    rnd = RoundRect(cx + pad, cy + pad, size, size, int(size / 5), stroke = 1,
                    fill=yellow, outline = yellow)

    group.append(rect)
    group.append(tri)
    group.append(circ)
    group.append(rnd)
    rect.fill = None
    tri.fill = None
    circ.fill = None
    rnd.fill = None

    time.sleep(2)

    rect.fill = red
    tri.fill = green
    circ.fill = blue
    rnd.fill = yellow
    time.sleep(2)
    clean_up(group)
```

```
    del rect
    del tri
    del circ
    del rnd
    gc.collect()

def sine_chart():
    gc.collect()
    cx = int(display.width / 2)
    cy = int(display.height / 2)
    minor = min(cx, cy)
    major = max(cx, cy)

    group.append(Line(cx, 0, cx, display.height, blue)) # v
    group.append(Line(0, cy, display.width, cy, blue)) # h

    for i in range(10):
        _n = simpleio.map_range(i, 0, 10, 0, major - 1)
        n = int(_n)
        group.append(Line(cx - n, cy - 5, cx - n, (cy - 5) + 11, blue)) # v
        group.append(Line(cx + n, cy - 5, cx + n, (cy - 5) + 11, blue)) # v
        group.append(Line(cx - 5, cy - n, (cx - 5) + 11, cy - n, blue)) # h
        group.append(Line(cx - 5, cy + n, (cx - 5) + 11, cy + n, blue)) # h

    for x in range(display.width):
        y = cy - int(math.sin((x - cx) * 0.05) * float(minor * 0.5))
        bitmap[x, y] = 1
    group.append(tile_grid)
    time.sleep(2)
    clean_up(group)

def widget0():
    gc.collect()
    data = [31, 42, 36, 58, 67, 88]
    num_points = len(data)

    text_area = label.Label(terminalio.FONT, text="Widget Sales", color=white)
    text_area.anchor_point = (0.5, 0.0)
    text_area.anchored_position = (display.width / 2, 3)
    group.append(text_area)
    for i in range(11):
        _x = simpleio.map_range(i, 0, 10, 0, display.width - 1)
        x = int(_x)
        group.append(Line(x, 20, x, display.height, blue))
        _y = simpleio.map_range(i, 0, 10, 20, display.height - 1)
        y = int(_y)
        group.append(Line(0, y, display.width, y, blue))
    prev_x = 0
    _prev_y = simpleio.map_range(data[0], 0, 100, display.height - 1, 20)
    prev_y = int(_prev_y)
    for i in range(1, num_points):
        _new_x = simpleio.map_range(i, 0, num_points - 1, 0, display.width - 1)
        new_x = int(_new_x)
        _new_y = simpleio.map_range(data[i], 0, 100, display.height - 1, 20)
        new_y = int(_new_y)
        group.append(Line(prev_x, prev_y, new_x, new_y, aqua))
        prev_x = new_x
        prev_y = new_y

    for i in range(num_points):
        _x = simpleio.map_range(i, 0, num_points - 1, 0, display.width - 1)
        x = int(_x)
        _y = simpleio.map_range(data[i], 0, 100, display.height - 1, 20)
        y = int(_y)
        group.append(Circle(x, y, 5, fill=None, stroke = 2, outline = white))

    time.sleep(2)
    clean_up(group)
```

```
def widget1():
    gc.collect()
    data = [31, 42, 36, 58, 67, 88]
    num_points = len(data)
    bar_width = int(display.width / num_points) - 4
    x_mapped_w = display.width + 2
    h_mapped_h = display.height + 20

    text_area = label.Label(terminalio.FONT, text="Widget Sales", color=white)
    text_area.anchor_point = (0.5, 0.0)
    text_area.anchored_position = (display.width / 2, 3)
    group.append(text_area)
    for i in range(11):
        _y = simpleio.map_range(i, 0, 10, 20, display.height - 1)
        y = int(_y)
        group.append(Line(0, y, display.width, y, blue))
    for i in range(num_points):
        _x = simpleio.map_range(i, 0, num_points, 0, x_mapped_w)
        x = int(_x)
        _height = simpleio.map_range(data[i], 0, 100, h_mapped_h, 0)
        height = int(_height)
        group.append(vectorio.Rectangle(pixel_shader=palette, width=bar_width,
                     height=display.height + 1, x=x, y=height, color_index = 2))

    time.sleep(2)
    clean_up(group)

def text_align():
    gc.collect()
    TEXT = "hello world"

    text_area_top_left = label.Label(terminalio.FONT, text=TEXT, color=red)
    text_area_top_left.anchor_point = (0.0, 0.0)
    text_area_top_left.anchored_position = (0, 0)

    text_area_top_middle = label.Label(terminalio.FONT, text=TEXT, color=orange)
    text_area_top_middle.anchor_point = (0.5, 0.0)
    text_area_top_middle.anchored_position = (display.width / 2, 0)

    text_area_top_right = label.Label(terminalio.FONT, text=TEXT, color=yellow)
    text_area_top_right.anchor_point = (1.0, 0.0)
    text_area_top_right.anchored_position = (display.width, 0)

    text_area_middle_left = label.Label(terminalio.FONT, text=TEXT, color=green)
    text_area_middle_left.anchor_point = (0.0, 0.5)
    text_area_middle_left.anchored_position = (0, display.height / 2)

    text_area_middle_middle = label.Label(terminalio.FONT, text=TEXT, color=aqua)
    text_area_middle_middle.anchor_point = (0.5, 0.5)
    text_area_middle_middle.anchored_position = (display.width / 2, display.height /
2)

    text_area_middle_right = label.Label(terminalio.FONT, text=TEXT, color=blue)
    text_area_middle_right.anchor_point = (1.0, 0.5)
    text_area_middle_right.anchored_position = (display.width, display.height / 2)

    text_area_bottom_left = label.Label(terminalio.FONT, text=TEXT, color=purple)
    text_area_bottom_left.anchor_point = (0.0, 1.0)
    text_area_bottom_left.anchored_position = (0, display.height)

    text_area_bottom_middle = label.Label(terminalio.FONT, text=TEXT, color=pink)
    text_area_bottom_middle.anchor_point = (0.5, 1.0)
    text_area_bottom_middle.anchored_position = (display.width / 2, display.height)

    text_area_bottom_right = label.Label(terminalio.FONT, text=TEXT, color=white)
    text_area_bottom_right.anchor_point = (1.0, 1.0)
    text_area_bottom_right.anchored_position = (display.width, display.height)

    group.append(text_area_top_middle)
```

```
    group.append(text_area_top_left)
    group.append(text_area_top_right)
    group.append(text_area_middle_middle)
    group.append(text_area_middle_left)
    group.append(text_area_middle_right)
    group.append(text_area_bottom_middle)
    group.append(text_area_bottom_left)
    group.append(text_area_bottom_right)

    time.sleep(2)
    clean_up(group)

def custom_font():
    gc.collect()
    my_font = bitmap_font.load_font("/Helvetica-Bold-16.pcf")
    text_sample = "The quick brown fox jumps over the lazy dog."
    text_sample = "\n".join(wrap_text_to_lines(text_sample, 28))
    text_area = label.Label(my_font, text="Custom Font", color=white)
    text_area.anchor_point = (0.0, 0.0)
    text_area.anchored_position = (0, 0)

    sample_text = label.Label(my_font, text=text_sample)
    sample_text.anchor_point = (0.5, 0.5)
    sample_text.anchored_position = (display.width / 2, display.height / 2)

    group.append(text_area)
    group.append(sample_text)

    time.sleep(2)
    clean_up(group)

    del my_font
    gc.collect()

def bitmap_example():
    gc.collect()
    blinka_bitmap = displayio.OnDiskBitmap("/blinka_computer.bmp")
    blinka_grid = displayio.TileGrid(blinka_bitmap,
pixel_shader=blinka_bitmap.pixel_shader)
    gc.collect()
    group.append(blinka_grid)

    time.sleep(2)
    clean_up(group)

    del blinka_grid
    del blinka_bitmap
    gc.collect()

def sensor_values():
    gc.collect()
    text_x = "X: %d" % randint(-25, 25)
    text_y = "Y: %d" % randint(-25, 25)
    text_z = "Z: %d" % randint(-25, 25)
    x_text = label.Label(terminalio.FONT, text=text_x, color=red)
    x_text.anchor_point = (0.0, 0.0)
    x_text.anchored_position = (2, 0)
    y_text = label.Label(terminalio.FONT, text=text_y, color=green)
    y_text.anchor_point = (0.0, 0.0)
    y_text.anchored_position = (2, 10)
    z_text = label.Label(terminalio.FONT, text=text_z, color=blue)
    z_text.anchor_point = (0.0, 0.0)
    z_text.anchored_position = (2, 20)
    group.append(x_text)
    group.append(y_text)
    group.append(z_text)

    for i in range(40):
        if i == 10:
```

```
            group.scale = 2
        elif i == 20:
            group.scale = 3
        elif i == 30:
            group.scale = 4
        x_text.text = "X: %d" % randint(-50, 50)
        y_text.text = "Y: %d" % randint(-50, 50)
        z_text.text = "Z: %d" % randint(-50, 50)
        time.sleep(0.1)
    time.sleep(0.1)
    clean_up(group)
    group.scale = 1

display.root_group = group

while True:
    show_shapes()
    sine_chart()
    widget0()
    widget1()
    text_align()
    custom_font()
    bitmap_example()
    sensor_values()
```

This example is a port of the Arduino **16bit_hello** code written by Phil B (https://adafru.it/18Bf). with some slight variation to show off some of the unique abilities of **displayio**.

The example begins by showing a rectangle, circle, triangle and rounded rectangle and changing the fill attribute from `None` to a color.

Then, a few chart variations are shown, including a sine wave pattern, line graph and bar graph.

Next is a text alignment example, showing how to use the `anchor_point` and `anchor_position` functions in the **adafruit_display_text** library.

Following that is a custom text example, loading a bitmap font instead of the built-in **terminalio** font.

Then there is a quick break from fonts to show off a bitmap image, specifically Blinka happily using her computer.

Finally, an example shows how to update the text in a `Label` object for projects where you want to display text information that updates over time.

Python Docs

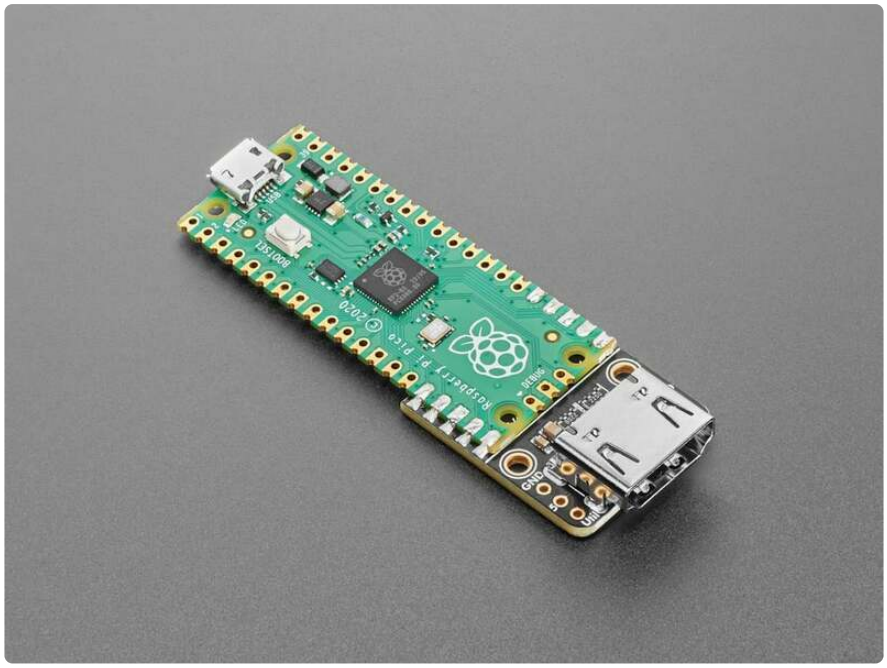
Python Docs (https://adafru.it/18Eu)

Arduino

Using the DVI Sock with Arduino involves connecting the DVI Sock to a Pico board, connecting the boards to an HDMI monitor and USB power, installing the Adafruit fork of the PicoDVI library and running the provided example code.

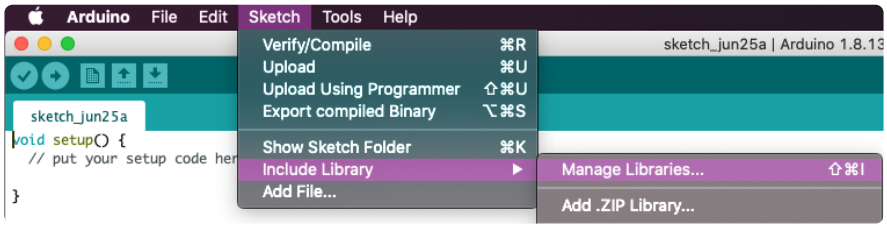
Wiring

Attach the Sock to the bottom pins of your Pico board, or solder it directly onto the 'end' like a li'l PCB sock.

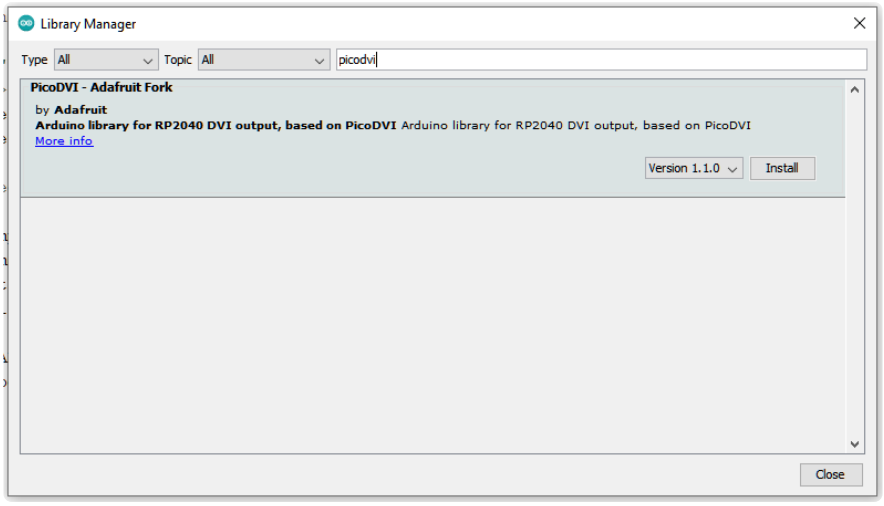


Library Installation

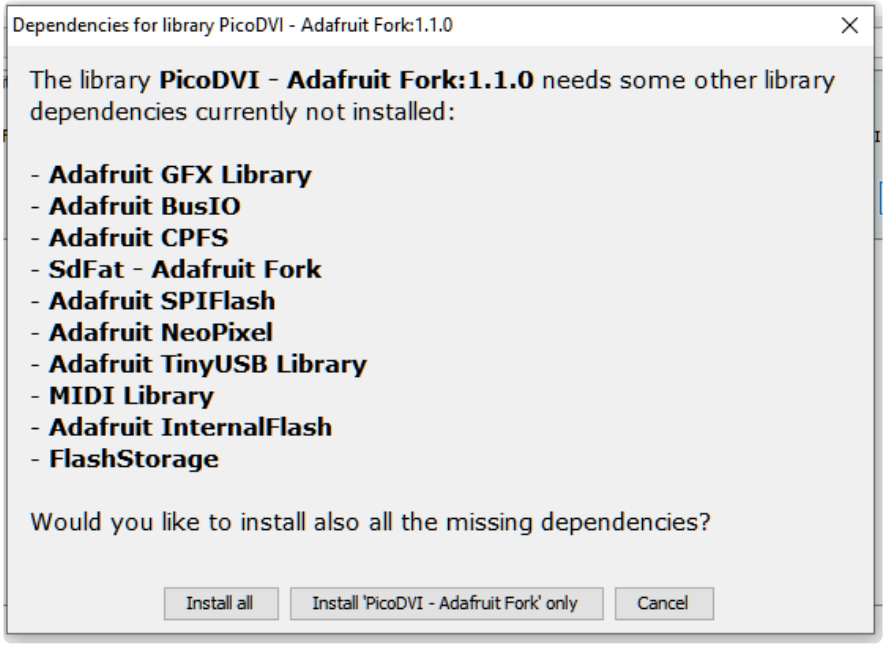
You can install the **Adafruit fork of the PicoDVI** library for Arduino using the Library Manager in the Arduino IDE.



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



If asked about dependencies for any of the libraries, 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: 2023 Phil B. for Adafruit Industries
// SPDX-License-Identifier: MIT
```

```
// Basic full-color PicoDVI test. Provides a 16-bit color video framebuffer to
// which Adafruit_GFX calls can be made. It's based on the EYESPI_Test.ino sketch.

#include <PicoDVI.h>                  // Core display & graphics library
#include <Fonts/FreeSansBold18pt7b.h> // A custom font

// Here's how a 320x240 16-bit color framebuffer is declared. Double-buffering
// is not an option in 16-bit color mode, just not enough RAM; all drawing
// operations are shown as they occur. Second argument is a hardware
// configuration -- examples are written for Adafruit Feather RP2040 DVI, but
// that's easily switched out for boards like the Pimoroni Pico DV (use
// 'pimoroni_demo_hdmi_cfg') or Pico DVI Sock ('pico_sock_cfg').
DVIGFX16 display(DVI_RES_320x240p60, pico_sock_cfg);

// A 400x240 mode is possible but pushes overclocking even higher than
// 320x240 mode. SOME BOARDS MIGHT SIMPLY NOT BE COMPATIBLE WITH THIS.
// May require selecting QSPI div4 clock (Tools menu) to slow down flash
// accesses, may require further over-volting the CPU to 1.25 or 1.3 V.
//DVIGFX16 display(DVI_RES_400x240p60, adafruit_feather_dvi_cfg);

void setup() { // Runs once on startup
  if (!display.begin()) { // Blink LED if insufficient RAM
    pinMode(LED_BUILTIN, OUTPUT);
    for (;;) digitalWrite(LED_BUILTIN, (millis() / 500) & 1);
  }
}

#define PAUSE 2000  // Delay (milliseconds) between examples
uint8_t rotate = 0; // Current screen orientation (0-3)
#define CORNER_RADIUS 0

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();
  show_canvas();

  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 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)) 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);
  }
```

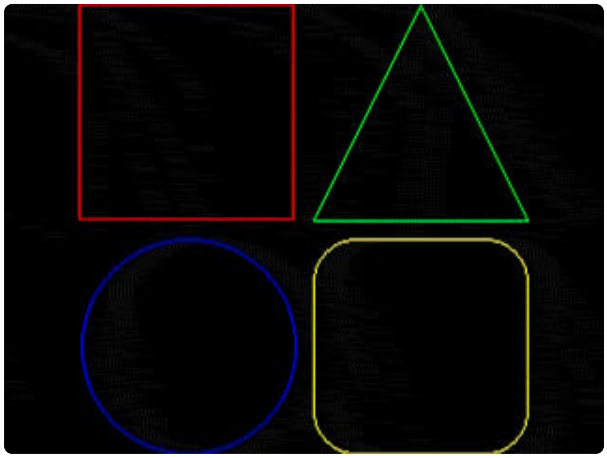
```
  delay(PAUSE);
} // END CUSTOM FONT EXAMPLE

// 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

  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 Example Code to the Pico. Then, you can connect the DVI Sock to an HDMI monitor and USB power to the Pico. You'll see the **16bit_hello** example display on the screen.

There is an excellent explainer page (https://adafru.it/18Ex) for the example code in the PicoDVI Arduino Library Learn Guide.



PicoDVI Arduino Library: Video Out for RP2040 Boards
By Phillip Burgess
16bit_hello

https://learn.adafruit.com/picodvi-arduino-library-video-out-for-rp2040-boards/16bit_hello

Arduino Docs

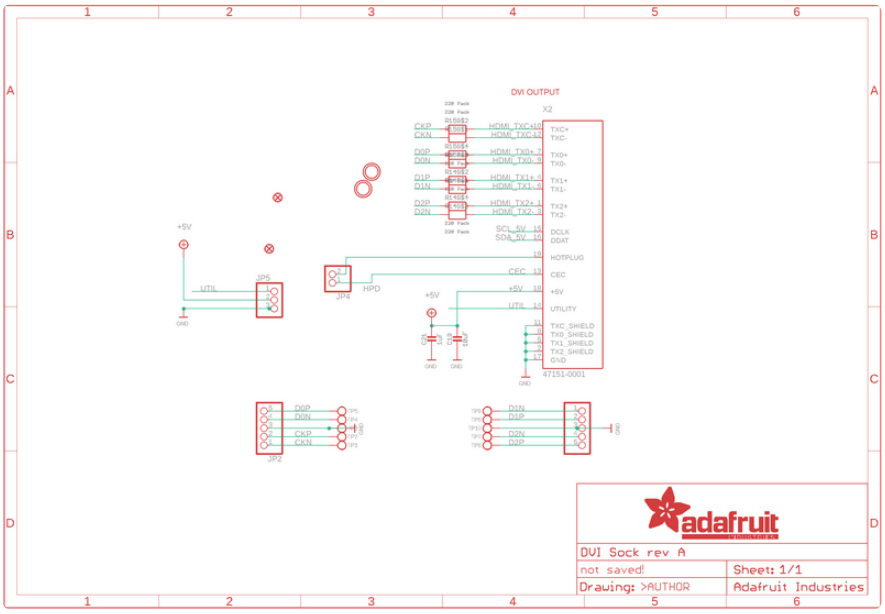
Arduino Docs (https://adafru.it/18Az)

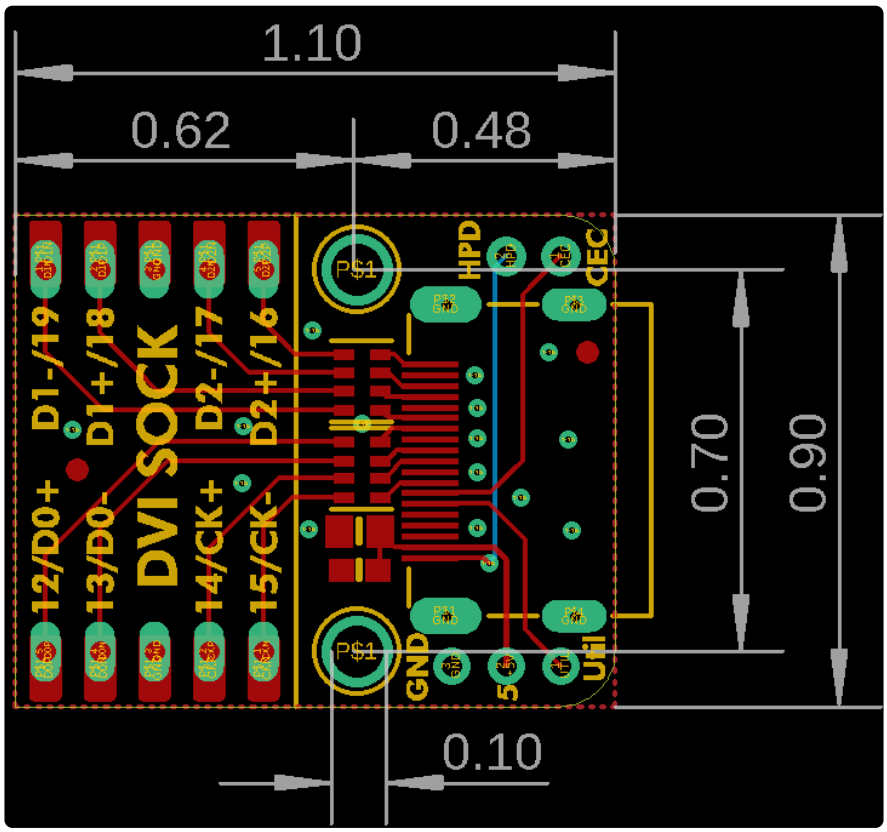
Downloads

Files

- HDMI Connector pinout (https://adafru.it/ZZB)
- EagleCAD PCB Files on GitHub (https://adafru.it/19Wd)
- Fritzing object in the Adafruit Fritzing Library (https://adafru.it/19We)

Schematic and Fab Print





Mouser Electronics

Authorized Distributor

Click to View Pricing, Inventory, Delivery & Lifecycle Information:

Adafruit:

5957