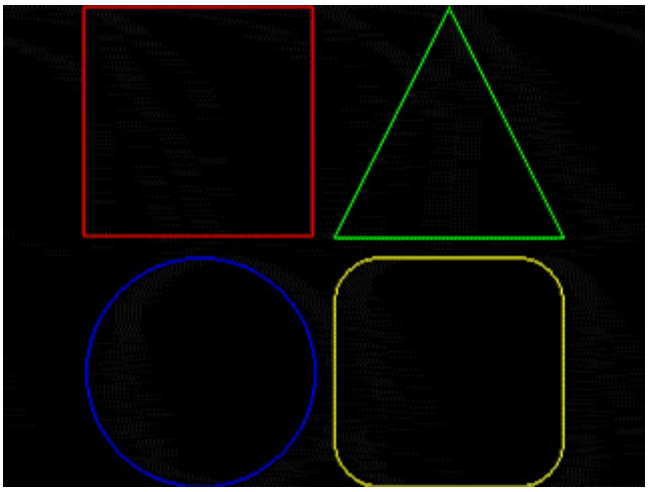
PicoDVI Arduino Library: Video Out for RP2040 Boards

Created by Phillip Burgess

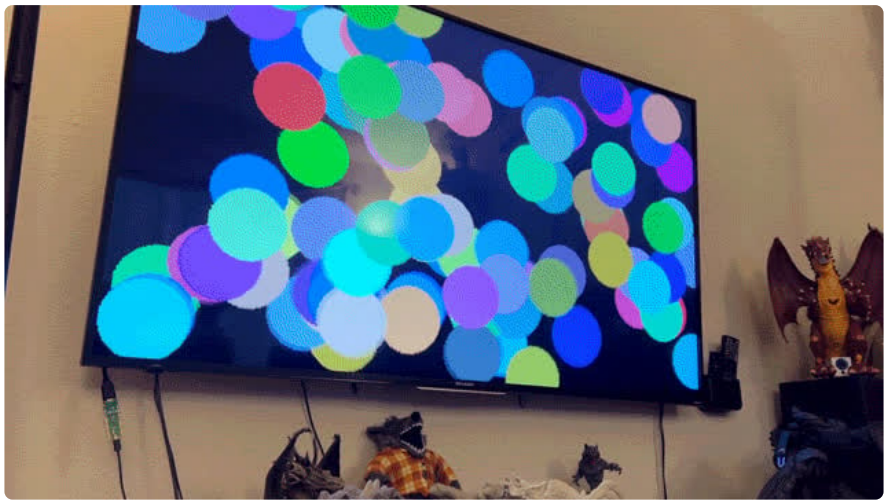


https://learn.adafruit.com/picodvi-arduino-library-video-out-for-rp2040-boards

Last updated on 2024-03-26 10:36:28 AM EDT

Table of Contents

Overview



IMPORTANT: the software documented here pushes the RP2040 microcontroller far beyond its design specifications. Just like PC overclocking, there's some risk of reduced component lifespan, though the extent (if any) can't be precisely quantified and could vary from one chip to another. Proceed at your own discretion.

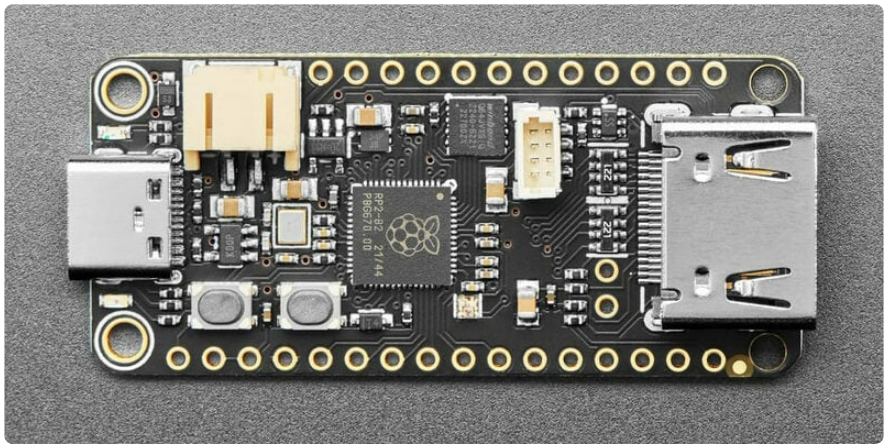
PicoDVI is a remarkable project by Luke Wren of the Raspberry Pi Foundation, capable of squeezing **digital video** directly from an RP2040 microcontroller. Imagine the creative applications for having these inexpensive devices linked to a display...not a tiny LCD, but the big bright screen in your living room or connected to your computer!

In its natural habitat, PicoDVI works with the Raspberry Pi Pico C/C++ SDK — a command-line development system — and projects using it involve careful planning and work to generate an image. What we've done is wrap up PicoDVI in an Arduino-flavoured wrapper: programmed within the familiar **Arduino IDE**, using graphics operations from the **Adafruit_GFX** library — draw lines, draw circles, some text and so forth. Novice coders can then develop video-based projects with minimal fuss.

Compatible Hardware

- PicoDVI can work on most **RP2040**-based microcontroller boards. It uses features unique to this chip and will not work anywhere else, no ESP32, no SAMD, etc.
- **8 GPIO pins** are used for output. They don't need to all be consecutive, but does require four pairs of consecutive pins.
- An **HDMI* breakout** board can be used, or a bare HDMI connector and some passive components. Or **better yet...**

- Several boards now have this all **baked in:** Adafruit's Feather RP2040 DVI (http://adafru.it/5710) (shown below), Pi Cowbell DVI, Pimoroni's Pico DV Demo Base (http://adafru.it/5674), or Luke Wren's own Pico DVI Sock (https://adafru.it/18Ax) design.



To perform its magic, the PicoDVI code sometimes takes liberties with video timing. Most monitors can successfully sync with this, but there's no guarantee that every monitor can latch on to every PicoDVI mode. You will need to experiment.

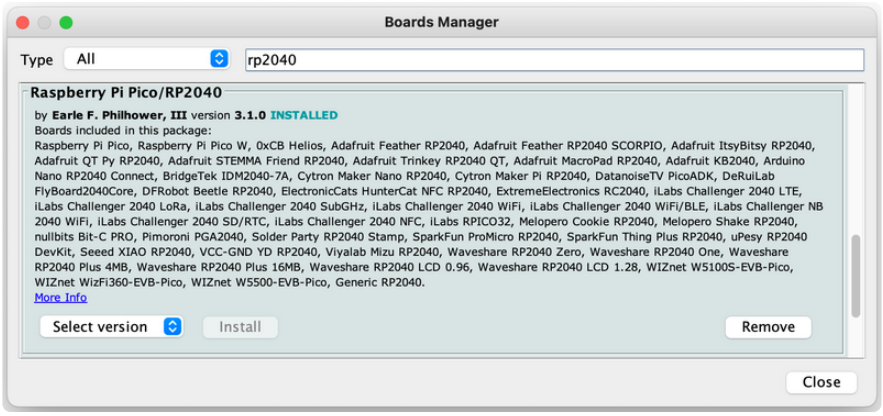
* Although an HDMI connector and cable are involved, strictly speaking this is DVI video. True and Proper HDMI™ is a proprietary, licensable format bringing together several technologies — video, audio, content protection, sometimes networking. HDMI's baseline non-content-protected video format happens to use DVI-compatible signaling, which is what the library generates. Nobody will fault you for casually calling it "HDMI out" (do younger folks even remember DVI monitors?), but if you're making a product or publishing a tutorial, best to say "DVI" to avoid potential licensing drama.

To reiterate an opening point: the PicoDVI code works by extreme overclocking. We've yet to witness failures from this, but it is a possibility. Thankfully the chips are affordable...the assumed risk is nowhere near the magnitude of overclocking a high-end PC CPU or graphics card.

Installation

PicoDVI relies on the **Earle Philhower III Arduino core** for programming — an optional package that makes most RP2040 boards work in the Arduino environment. If you've previously followed any guides for our RP2040-based boards, you likely already have this installed...just check that you're up to date with the latest (**3.1.0 or newer**).

If that sounds unfamiliar, this guide walks through the process (https://adafru.it/18Ay).



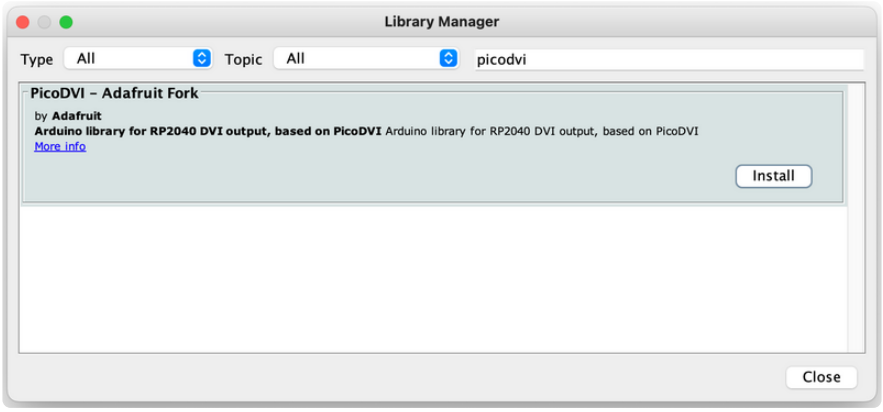
Once installed, the Arduino IDE **Tools→Board** menu will include a rollover for "Raspberry Pi RP2040 Boards," and you can find and select whatever board type you're using (e.g. Feather RP2040 DVI or Raspberry Pi Pico).

Next, the **PicoDVI library** can be installed from the **Arduino Library Manager**. From the Sketch menu...

Sketch→Include Library→Manage Libraries...

Enter "**picodvi**" in the search field and look for **PicoDVI - Adafruit Fork** in the results. Click "Install," then "Close."

Our version of PicoDVI depends on the **Adafruit_GFX** library. The Library Manager should install this automatically if not already present, but if using an older version of the Arduino IDE you might need to search for and install it manually.



This is our "fork" of the original PicoDVI project, meaning as much of the original code is preserved with minimal changes. What we've done is add an Arduino-compliant C++ wrapper to make this command-line library work with the friendlier Arduino IDE, and implemented simple raster framebuffers for drawing. All the original stuff is there if you want to dig in and learn, though the original examples as written won't build in the Arduino IDE.

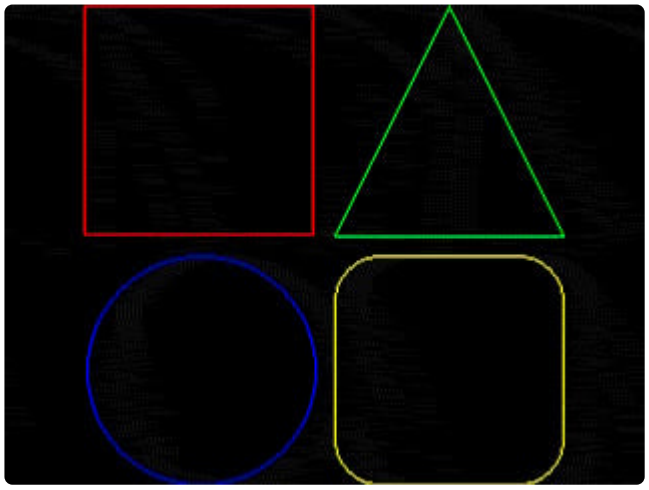
Here's our fork on GitHub (https://adafru.it/18Az), and Luke Wren's original project (https://adafru.it/ZTf).

Examples

We've created several examples to showcase the library in use. Each tends to highlight one new capability of PicoDVI, none too complex or technical. With core and library installed, you'll find these in the **Files→Examples→PicoDVI - Adafruit Fork** menu in alphabetical order, but let's walk through from simple to fancy. Even if you don't plan on trying any or all of these examples, please skim through, some key concepts are explained...

16bit_hello

Please read through at least this example. Most of the concepts covered here apply to all the examples and to using the PicoDVI library in general, and it would be a tedious read with all these details reiterated on every page.



Although this example is a lot of code, it's also the most familiar if you've previously used the Adafruit_GFX library with any of our color TFT or OLED displays...in fact, it's the same demo we now use for all our EYESPI screens (https://adafru.it/18fg), with only a slight adjustment for DVI output. If you haven't used Adafruit_GFX, we have a whole guide explaining the groundwork (https://adafru.it/doL).

The GFX parts are well-commented and we won't go through the whole example here, just mentioning the most important lines in this example that set it apart from the usual TFT projects...

Near the very top of the code:

```
#include &lt;PicoDVI.h&gt;
```

This header file establishes all that needs known about the PicoDVI library. In projects that use TFT displays, the equivalent line might reference `Adafruit_ST7789.h` or `Adafruit_ILI9341.h` — headers for different display controllers — and it's the same idea here, but for the DVI display.

A little further down, a global declaration is made before the `setup()` function:

```
DVIGFX16 display(DVI_RES_320x240p60, adafruit_feather_dvi_cfg);
```

This creates a global object, `display` (it's global — usable anywhere in the sketch — because it's declared outside any function), of type `DVIGFX16`, which is a **16-bit color framebuffer** (a section of memory dedicated for graphical display). 16-bit colors are described further in the GFX tutorial (https://adafru.it/JD8). The two arguments passed to this function define the resolution and what GPIO pins are used...more on those in a moment, but first...

Then, inside the `setup()` function (which runs once at startup), there's a call like this:

```
display.begin()
```

`display` is the previously-declared global object, `.begin()` is the function that actually starts up the DVI output (without this, you'll just get a No Signal message on most monitors).

In the example, you'll notice the **return value is tested**...and if **false**, the program just stops there, blinking the LED to indicate there's a problem. This can happen due to a lack of resources...usually insufficient RAM or other finite resources of the chip (three of the chip's four PIO state machines are needed for DVI).

While the RP2040 chip has a generous 264 kilobytes of RAM, a full-color screenful like this uses a majority of it...about **150 KB** (320 × 240 × 2 bytes/pixel). On a typical TFT or OLED display, that would be offloaded to a display controller chip, but with DVI it occupies main RAM. The tradeoff is stunning performance...toggling bits in memory is orders of magnitude faster than pushing pixels even over a fast SPI connection.

Let's look at the constructor again, that global declaration for the display, and explore the arguments passed:

```
DVIGFX16 display(DVI_RES_320x240p60, adafruit_feather_dvi_cfg);
```

Looking at the second argument first, simpler to explain — `adafruit_feather_dvi_cfg` — this is the name of a structure tucked away in the header files, and configures the code to work on the **Adafruit Feather RP2040 DVI** board. There are others! You might change this to `pimoroni_demo_hdmi_cfg` if using a Pimoroni Pico DV Demo Base board, or `pico_sock_cfg` for a Pico DVI sock. Each of these boards adopted a different set of DVI output pins in their design. (The complete set of ready-made pin configurations can be found in the library file PicoDVI/software/include/common_dvi_pin_configs.h — or, if using an HDMI breakout and your own custom pinout, declare a `dvi_serialiser_cfg` structure of your own and pass it to the constructor).

The first argument, `DVI_RES_320x240p60`, selects the output resolution and refresh rate: 320×240 pixels at 60 Hz. Only **a very limited set of resolutions are supported**, and this is one of them. 320×240 is a 4:3 aspect ratio and might look best on an older DVI monitor, or with "pillarboxing" (left and right black bars) enabled on a more modern 16:9 HDMI monitor...most will have a menu selection somewhere for pillarboxing or if you'd prefer the image "stretched" to the full display width...pixels are then rectangular rather than square, but it does use the whole display.

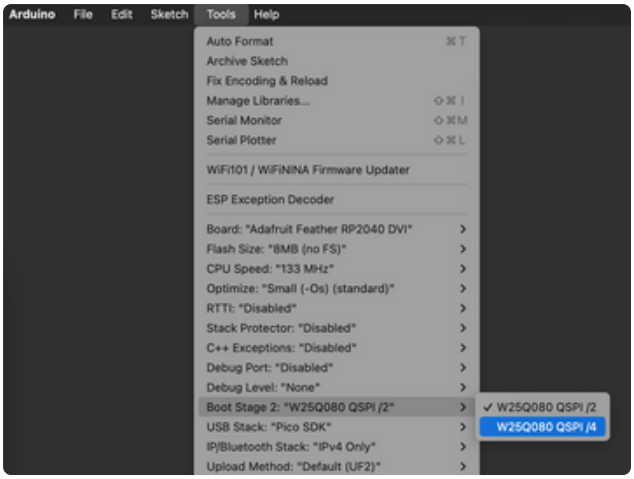
Some monitors will report this as 640×480 resolution, but it's really only half that. The library is doubling up pixels on both the horizontal and vertical axes, and there wouldn't be enough RAM to buffer a full image at the larger size anyway.

There's a line, commented out by default, that selects a wider mode. You can enable this (and comment out the first one instead)...but read on for extra steps this entails:

```
DVIGFX16 display(DVI_RES_400x240p60, adafruit_feather_dvi_cfg);
```

This is closer to 16:9 aspect ratio (not exactly, but less visibly stretched) at the expense of some extra RAM...now using 192K, leaving only about a Commodore 64's worth for your own project to do its thing.

BEFORE YOU ATTEMPT THIS RESOLUTION THOUGH, there's a compile-time setting that MUST be manually enabled:



With most Adafruit RP2040 boards, including the Feather DVI, access to flash memory must be explicitly slowed down... the CPU would otherwise outpace it and lock up, it's running so fast in this mode!

From the Arduino **Tools** menu, look for the **Boot Stage 2** rollover and select **W25Q080 QSPI/4**.

This is wild frontier engineering, running the RP2040 at an absolutely bonkers clock rate, and might not even work across every chip. And when it doesn't, locks up so hard that not even the LED error blink will happen, and your computer may get slow and choppy as it tries to understand this weird thing connected to USB. **If this happens, unplug the board, hold down the BOOT button while re-connecting, and set the code back to the milder 320×240 resolution.** Sorry about that.

There's a secret optional third argument to the constructor that can run the RP2040 chip at a slightly higher internal voltage. Extreme overclocking (such as at higher resolutions) may require this. Normally the chip operates internally at 1.1 Volts DC, and by default the library boosts this slightly to 1.2V. That's beyond the chip's "official" rated specification, but just like PC overclocking, can keep the system more stable. (Or, if you find the approved 1.1V sufficiently stable, can select that.) Just tack it on as an extra argument like so:

```
DVIGFX16 display(DVI_RES_400x240p60, adafruit_feather_dvi_cfg, VREG_VOLTAGE_1_25);
```

The available voltage selections include:

`VREG_VOLTAGE_0_85` (0.85 Volts)
`VREG_VOLTAGE_0_90` (0.9V)
`VREG_VOLTAGE_0_95` (0.95V)
`VREG_VOLTAGE_1_00` (1.0V)
`VREG_VOLTAGE_1_05` (1.05V)
`VREG_VOLTAGE_1_10` **(1.1V, normal system default)**
`VREG_VOLTAGE_1_15` (1.15V)
`VREG_VOLTAGE_1_20` **(1.2V, library default if not specified)**
`VREG_VOLTAGE_1_25` (1.25V)
`VREG_VOLTAGE_1_30` (1.3V, maximum possible from internal regulator)

Over-volting does carry with it associated risks, as mentioned on the opening page.

Once everything is set up, the code then goes into some straightforward GFX demos, just as it would with a TFT or OLED display...

Source Code

```
// 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, adafruit_feather_dvi_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,
  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
```

8bit_single_buffer and 8bit_double_buffer

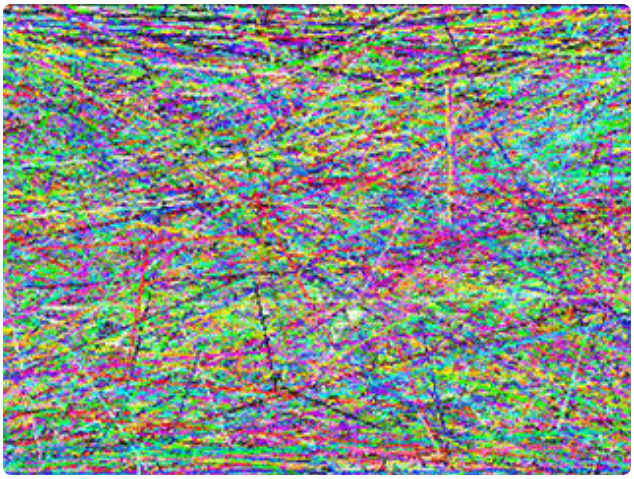
The prior 16-bit example laid the groundwork and might be all you need as a starting point for most projects, especially for rich full-color graphics. But it does have some drawbacks...

- The full-color framebuffer consumes over half of available RAM. What's left over is often sufficient for some good projects, but really complex tasks might suffer.
- Graphics in motion will likely flicker and/or exhibit tearing — where a moving object is briefly seen half in its old and new positions, because screen and microcontroller aren't synchronized.

The library offers solutions for each:

- An **8-bit color** mode uses just one byte per pixel instead of two; a 320×240 pixel occupies half the space, about 75K now. The tradeoff is that a maximum of 256 colors can be used.
- A **double-buffered** 8-bit mode prevents flicker and tearing by maintaining two screen buffers: one currently shown on the monitor, and a second where drawing occurs "in the background," which can then alternate (synchronized to the display refresh, so no tearing). Tradeoff is that this uses just as much RAM as a full 16-bit screen.

The 8-bit examples are much shorter than the prior one. They're not here to demonstrate every kitchen-sink feature of Adafruit_GFX, simply to show what's unique to 8-bit modes...



8bit_single_buffer

This starts out the same as before, by #including the `PicoDVI.h` header file. And then the global `display` declaration is ever-so-slightly different:

```
DVIGFX8 display(DVI_RES_320x240p60, false, adafruit_feather_dvi_cfg);
```

It's now a `DVIGFX8` object (rather than `DVIGFX16`), and there's one extra argument in the middle there. `false` here tells the library to use single-buffered mode, the more memory-friendly option described above. Double-buffered mode is covered later.

All the other constructor arguments work the same as with `DVIGFX16` explained on the prior page; resolution (including the need for special flash memory settings in "wide" mode), pinout and an optional voltage selection (not specified here, using the default).

`DVIGFX8` allows up to 256 colors, but you get to decide which 256 colors from the full 16-bit color space. This is done through the `setColor()` function, which accepts a color index (0 to 255) and a 16-bit "RGB565" color. By default, all 256 colors are set to 0 (black)...you need to set up something for your program. This first example leaves color 0 (the default background color) unchanged — black — then fills slots 1–254 with random 16-bit values, and finally the last slot (255) with white (0xFFFF). There's no telling what the random colors might end up being, but at least this way we know there's two contrasting colors at the very ends:

```
for (int i=1; i&lt;255; i++) display.setColor(i, random(65536));
display.setColor(255, 0xFFFF);
```

The `loop()` function then calls the usual Adafruit_GFX `drawLine()` function to draw lines between random start and end points...and each with a random **color index from 0 to 255** (corresponding to the table initialized with setColor() earlier) rather than a 16-bit RGB565 color.

```
display.drawLine(random(display.width()), random(display.height()),
                 random(display.width()), random(display.height()),
                 random(256));
```

This illustrates the speed of PicoDVI versus an SPI-attached TFT display...lines draw so fast the screen almost resembles pure noise.

8bit_double_buffer

This example shows how to create flicker-free, tear-free animation, moving dozens of objects about the screen.

Code starts out the same as the prior example, with the small change of passing `true` as the second argument to the constructor:

```
DVIGFX8 display(DVI_RES_320x240p60, true, adafruit_feather_dvi_cfg);
```

By enabling double buffering, this now uses just as much RAM as 16-bit color mode (about 150K, or 187K for widescreen) but permits buttery smooth effects.

As in the prior example, a random color palette is created, with minor changes this time. Black and white are still used for entries 0 and 255, but for anything in-between we'd like slightly brighter colors (25–100% rather than 0–100%)...and, rather than a single packed 16-bit value, we have the option of calling `setColor()` with three 8-bit values (0–255):

```
display.setColor(i+1, 64 + random(192), 64 + random(192), 64 + random(192));
```

Although this variant of `setColor()` accepts three 8-bit inputs, the stored color is not really 24-bit. Behind the scenes, the values are quantized to 5 or 6 bits and packed together into a 16-bit value. It's offered this way because a whole lot of existing graphics code — and color pickers in programs such as Photoshop — rely extensively on 24-bit colors, not 16-bit. You can use whatever's easiest.

Skipping ahead to the `loop()` function for a moment...for each frame of animation, the program:

- Clears the entire screen
- Draws a number of filled circles
- Updates the positions of those circles ("bouncing balls") for the next frame

It doesn't even bother carefully erasing everything in the old positions and then drawing new ones...the whole screen is just cleared and drawn anew. Filling the screen is super fast and the code is much simpler this way.

You don't see any flicker when the screen is cleared and the balls are drawn, because of one last step:

```
display.swap();
```

All that clearing and drawing takes place in a background framebuffer, out of sight. The call to `swap()` brings that buffer to the front (synchronized to the next video refresh), and the front buffer to the back for subsequent drawing operations.

`swap()` can optionally accept two boolean arguments, and backing up to the end of the `setup()` function now, you can see that in action:

```
display.swap(false, true);
```

The first argument, if `true`, will (after the video frame sync where the swap takes place) copy the new foreground buffer contents to the background buffer. This can be helpful for programs that generate animation incrementally; just drawing any changes over the prior frame. If `false`, no copy is performed...ideal for situations like this bouncing ball example, where the whole buffer is simply cleared and drawn anew.

The second argument does the same thing for the color palette, which is independent for the front and back buffers. Sometimes programs simulate animation through "color cycling" — not redrawing any pixels, but just moving a set of colors through a loop. This too would exhibit "tearing" if not for double-buffering. The bouncing balls demo doesn't do color cycling, but it does call `swap(false, true)` once after colors are initialized, so both the front and back buffers are using the same palette.

Source Code: 8bit_single_buffer

```
// 8-bit Adafruit_GFX-compatible framebuffer for PicoDVI.
```

```
#include <PicoDVI.h>

// Here's how a 320x240 8-bit (color-paletted) framebuffer is declared.
// Second argument ('false' here) means NO double-buffering; all drawing
// operations are shown as they occur. Third 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').
DVIGFX8 display(DVI_RES_320x240p60, false, adafruit_feather_dvi_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.
//DVIGFX8 display(DVI_RES_400x240p60, false, 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);
  }

  // Randomize color palette. First entry is left black, last is set white.
  for (int i=1; i<255; i++) display.setColor(i, random(65536));
  display.setColor(255, 0xFFFF);
}

void loop() {
  // Draw random lines
  display.drawLine(random(display.width()), random(display.height()),
random(display.width()), random(display.height()), random(256));
}
```

Source Code: 8bit_double_buffer

```
// Double-buffered 8-bit Adafruit_GFX-compatible framebuffer for PicoDVI.
// Animates without redraw flicker. Requires Adafruit_GFX >= 1.11.4

#include <PicoDVI.h>

// Here's how a 320x240 8-bit (color-paletted) framebuffer is declared.
// Second argument ('true' here) enables double-buffering for flicker-free
// animation. Third 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').
DVIGFX8 display(DVI_RES_320x240p60, true, adafruit_feather_dvi_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.
//DVIGFX8 display(DVI_RES_400x240p60, true, adafruit_feather_dvi_cfg);


#define N_BALLS 100 // Number of bouncy balls to draw, 1-254 (not 255)
struct {
  int16_t pos[2]; // Ball position (X,Y)
  int8_t  vel[2]; // Ball velocity (X,Y)
} ball[N_BALLS];

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);
```

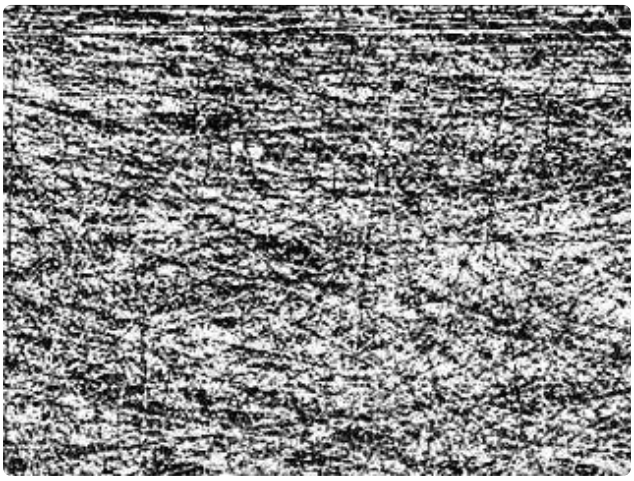
```
  }

  // Randomize initial ball positions, velocities and colors
  for (int i=0; i<N_BALLS; i++) {
    display.setColor(i+1, 64 + random(192), 64 + random(192), 64 + random(192));
    ball[i].pos[0] = 10 + random(display.width() - 20);
    ball[i].pos[1] = 10 + random(display.height() - 20);
    do {
      ball[i].vel[0] = 2 - random(5);
      ball[i].vel[1] = 2 - random(5);
    } while ((ball[i].vel[0] == 0) && (ball[i].vel[1] == 0));
  }
  display.setColor(255, 0xFFFF); // Last palette entry = white
  display.swap(false, true); // Duplicate same palette into front & back buffers
}

void loop() {
  // Clear back framebuffer and draw balls (circles) there.
  display.fillScreen(0);
  for (int i=0; i<N_BALLS; i++) {
    display.fillCircle(ball[i].pos[0], ball[i].pos[1], 20, i + 1);
    // After drawing each one, update positions, bounce off edges.
    ball[i].pos[0] += ball[i].vel[0];
    if ((ball[i].pos[0] <= 0) || (ball[i].pos[0] >= display.width())) ball[i].vel[0]
*= -1;
    ball[i].pos[1] += ball[i].vel[1];
    if ((ball[i].pos[1] <= 0) || (ball[i].pos[1] >= display.height()))
ball[i].vel[1] *= -1;
  }
  // Swap front/back buffers, do not duplicate current screen state to next frame,
  // we'll draw it new from scratch each time.
  display.swap();
}
```

1bit_single_buffer and 1bit_double_buffer

Sometimes a sharp image is more important that a variety of colors. PicoDVI has a setting offering double the resolution on each axis — 640×480, or 800×480 in wide mode — if you're fine with **strictly black-and-white** graphics. Even with all the extra pixels, this uses only half the RAM of the 8-bit modes, or one fourth the 16-bit mode: about 38K for single-buffered 640×480 graphics, 75K double-buffered (and 47 or 94K for the widescreen variants).



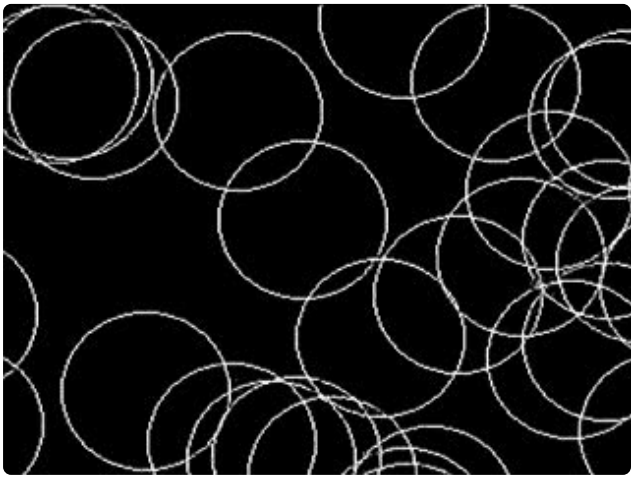
1bit_single_buffer

The global `display` declaration is very similar to the 8-bit version:

```
DVIGFX1 display(DVI_RES_640x480p60, false, adafruit_feather_dvi_cfg);
```

The display type is `DVIGFX1` now (instead of `DVIGFX8`), and the resolution doubled by requesting `DVI_RES_640x480p60`. Second argument is `false` because this example is single-buffered, and the remaining arguments (pinout and optional voltage setting) are the same as before. Remember that you may need to adjust the flash memory timing if using the widescreen `DVI_RES_800x480p60` mode.

The remaining code in `loop()` then just draws lines between random endpoints, randomly selecting color 0 (black) or color 1 (white). Any GFX function could be used here — circles, text (including fonts) and so forth — using just those two color values. That's all there is to it!



1bit_double_buffer

This is the bouncing-balls demo again, but strictly black-and-white. You can probably predict how this will all go...

```
DVIGFX1 display(DVI_RES_640x480p60, true, adafruit_feather_dvi_cfg);
```

Second argument is `true` now for double buffering, and the rest is all the same. No color palette is set up this time because 1-bit mode is always just black (0) or white (1).

`loop()` then works similarly to the 8-bit version: clear the whole screen, draw circles (outlined this time), update their positions for next time, and swap front/back buffers. `swap()` can optionally accept a single `true` / `false` argument (rather than two in the 8-bit case) whether to copy the front framebuffer contents to the back. It's not used in this case because the full screen is redrawn; `false` is implied if not provided.

Source Code: 1bit_single_buffer

```
// Simple 1-bit Adafruit_GFX-compatible framebuffer for PicoDVI.

#include <PicoDVI.h>

// Here's how a 640x480 1-bit (black, white) framebuffer is declared.
// Second argument ('false' here) means NO double-buffering; all drawing
// operations are shown as they occur. Third 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').
DVIGFX1 display(DVI_RES_640x480p60, false, adafruit_feather_dvi_cfg);

// An 800x480 mode is possible but pushes overclocking even higher than
// 640x480 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.
//DVIGFX1 display(DVI_RES_800x480p60, false, 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);
  }
}

void loop() {
  // Draw random lines
  display.drawLine(random(display.width()), random(display.height()), // Start X,Y
                   random(display.width()), random(display.height()), // End X,Y
                   random(2)); // Color (0 or 1)
}
```

Source Code: 1bit_double_buffer

```
// Double-buffered 1-bit Adafruit_GFX-compatible framebuffer for PicoDVI.
// Animates without redraw flicker. Requires Adafruit_GFX >= 1.11.5

#include <PicoDVI.h>

// Here's how a 640x480 1-bit (black, white) framebuffer is declared.
// Second argument ('true' here) enables double-buffering for flicker-free
// animation. Third 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').
DVIGFX1 display(DVI_RES_640x480p60, true, adafruit_feather_dvi_cfg);

// An 800x480 mode is possible but pushes overclocking even higher than
// 640x480 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.
//DVIGFX1 display(DVI_RES_800x480p60, true, adafruit_feather_dvi_cfg);

#define N_BALLS 100 // Number of bouncy balls to draw
struct {
  int16_t pos[2]; // Ball position (X,Y)
  int8_t  vel[2]; // Ball velocity (X,Y)
} ball[N_BALLS];

void setup() { // Runs once on startup
```

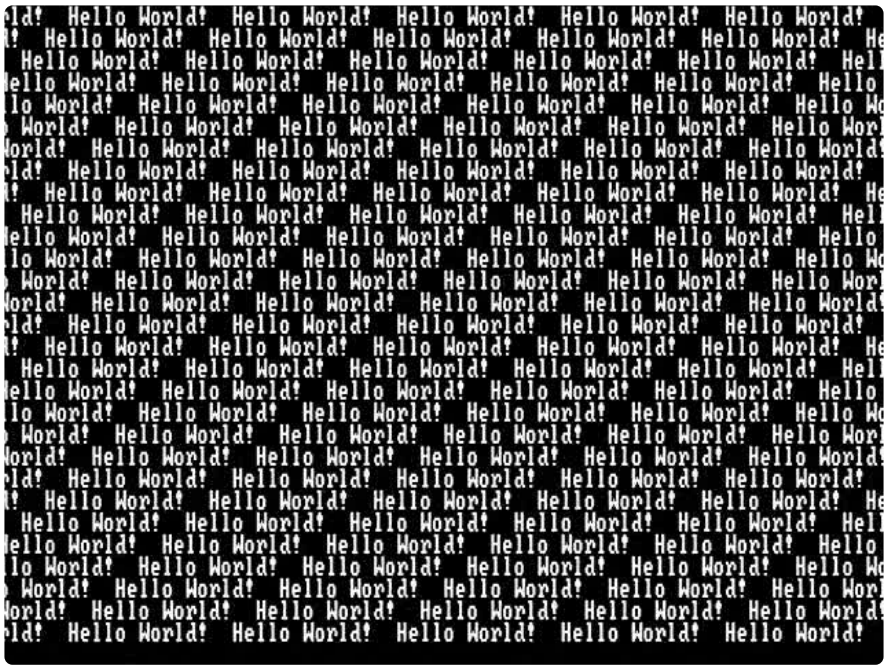
```
  if (!display.begin()) { // Blink LED if insufficient RAM
    pinMode(LED_BUILTIN, OUTPUT);
    for (;;) digitalWrite(LED_BUILTIN, (millis() / 500) & 1);
  }

  // Randomize initial ball positions and velocities
  for (int i=0; i<N_BALLS; i++) {
    ball[i].pos[0] = 10 + random(display.width() - 20);
    ball[i].pos[1] = 10 + random(display.height() - 20);
    do {
      ball[i].vel[0] = 4 - random(9);
      ball[i].vel[1] = 4 - random(9);
    } while ((ball[i].vel[0] == 0) && (ball[i].vel[1] == 0));
  }
}

void loop() {
  display.fillScreen(0); // Clear back framebuffer...
  // And draw bouncy balls (circles) there
  for (int i=0; i<N_BALLS; i++) {
    display.drawCircle(ball[i].pos[0], ball[i].pos[1], 40, 1);
    // After drawing each one, update positions, bounce off edges.
    ball[i].pos[0] += ball[i].vel[0];
    if ((ball[i].pos[0] <= 0) || (ball[i].pos[0] >= display.width())) ball[i].vel[0]
*= -1;
    ball[i].pos[1] += ball[i].vel[1];
    if ((ball[i].pos[1] <= 0) || (ball[i].pos[1] >= display.height()))
ball[i].vel[1] *= -1;
  }

  // Swap front/back buffers, do not duplicate current screen state to next frame,
  // we'll draw it new from scratch each time.
  display.swap();
}
```

1bit_text



Adafruit_GFX has some nice text and font support, but occasionally you may want something minimalist and fast, like the text mode of old PC's. PicoDVI offers a fixed-

width pure character mode that uses far less RAM than even the 1-bit graphics modes, and conveniently scrolls upward when text reaches the bottom.

Looking at the `display` declaration for this example:

```
DVItext1 display(DVI_RES_640x240p60, adafruit_feather_dvi_cfg);
```

The object type is now `DVItext1` (optimistically named with the "1" in case we add color text mode(s) in the future). The arguments should be familiar by now, but the resolution setting — `DVI_RES_640x240p60` — is a new one.

The character cells in this text mode are **8 by 8 pixels**. A **640** pixel width will thus yield **80 columns** of text. The peculiar **240** pixel height used here makes things vertic ally stretched; each character cell is tall and narrow, and this mode yields **80×30** characters overall.

The result looks incredibly similar to vintage 80 column PC text — in fact the shapes are based on the IBM VGA font. This is also known as the "Code Page 437 (https://adafru.it/Uby)" character set. Characters are 8 bits...0–255...there's a variety of symbols and accented letters, but no Unicode support, nor any alternate fonts.

A 100-column wide mode is possible (following the same compilation rules as explained in prior examples), and up to 60 rows if one foregoes the vertical stretch:

```
DVItext1 display(DVI_RES_800x480p60, adafruit_feather_dvi_cfg);
```

There's little to say here about use other than it supports the same `print()` and `println()` calls as Arduino Serial or Adafruit_GFX:

```
display.print("Hello World!  ");
```

It's pretty basic, there's no word wrap at the right edge...but when text reaches the bottom of the screen, everything will scroll upward automatically.

Source Code

```
// 1-bit (black, white) text mode for PicoDVI.

#include <PicoDVI.h>

// Here's how an 80x30 character display is declared. First argument,
// resolution, is full display pixel count...character cells are 8x8 pixels,
// yielding the 80x30 result. 640x240 uses "tall" pixels, the result of all
// this is very reminiscent of IBM VGA mode. 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').
DVItext1 display(DVI_RES_640x240p60, adafruit_feather_dvi_cfg);

// Wider and taller modes are possible. Wider pushes overclocking even
// higher than 640x480 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 over-volting the CPU to 1.25 or 1.3 V.
// Here's how a 100x60 char display might be declared:
//DVItext1 display(DVI_RES_800x480p60, adafruit_feather_dvi_cfg);

// A reduced refresh rate display doesn't as aggressive an over-clock
// This timing is verified to work on https://www.adafruit.com/product/2232
//DVItext1 display(DVI_RES_800x240p30, 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);
  }
}

void loop() {
  display.print("Hello World!  ");
  delay(50);
}
```

---

virtual_spitft

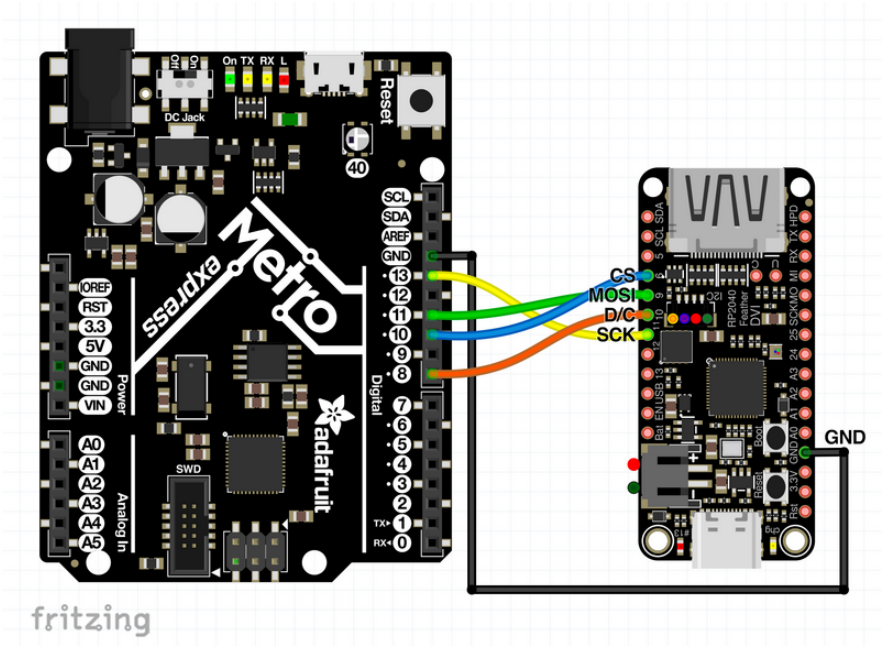
This is **not a self-contained example**. It works in conjunction with another microcontroller that thinks it's connected to an ST7789 or ILI9341 TFT display, issuing those graphics to the DVI output instead (or, if a TFT screen is present, it can display on both). This requires additional hardware and code…and it probably can't keep up with the fastest devices…but seems to work well enough with midrange parts like SAMD21 (e.g. Metro Express).

This taps into the **SPI output** of the host device and a couple extra control signals. These can be routed to most any input pins on the RP2040 running the PicoDVI code, but **three of these pins — SPI MOSI, TFT D/C, SPI CLK — must be sequential** (e.g. pins 9–11, or A1–A3, etc.). The fourth — TFT CS — can go to any available pin, and of course **ground should be connected between both boards**.

The input pins are defined near the top of the example code:

```
#define PIN_DATA 9 // 3 contiguous pins start here: data, DC, clk
#define PIN_CS   6 // Chip-select need not be contiguous
```

Here is one possible wiring setup, for the Adafruit_ST7789 graphicstest.ino example running on an Adafruit Metro M0 board:



The “host” device could be just about anything, really. A Metro M0 board was chosen because it uses 3.3V logic — it can connect directly to the Feather without a logic level shifter, making the diagram a little simpler.

The wires shown happen to be the defaults for the “graphicstest” example on that board, but in reality might connect elsewhere…you’d need to look through the code. Also, there could also be a TFT physically attached, with the signals split both to the TFT and the Feather RP2040 DVI board.

If the host device uses **5 Volt logic** (Metro Mini, Arduino Nano, etc.), a logic-level shifter such as the 74LVC245 IC (http://adafru.it/735) should be added between to bring these signals down to a safe 3.3V for the RP2040. This is not shown in the wiring diagram, but the connections can be easily derived from the chip’s datasheet (https://adafru.it/18AA). Vcc connects to 3.3V from the RP2040 board, OE to ground, and DIR to either 3.3V or ground depending which side you’re using for “in” vs. “out.”

This example is an interesting concept but not especially robust. The RP2040 board running PicoDVI should be powered on first, a fast host microcontroller might outpace the code running on the RP2040, and if the two get out of sync, it may or may not re-synchronize successfully.

Source Code

```
// PicoDVI-based "virtual SPITFT" display. Receives graphics commands/data
// over 4-wire SPI interface, mimicking functionality of displays such as
// ST7789 or ILI9341, but shown on an HDMI monitor instead.

#include <PicoDVI.h> // Core display & graphics library
```

```
// Configurables ----

// GPIO connected to (or shared with) TFT control.
// Careful not to overlap the DVI pins.
#define PIN_DATA 9 // 3 contiguous pins start here: data, DC, clk
#define PIN_CS   6 // Chip-select need not be contiguous

// 320x240 16-bit color display (to match common TFT display resolution):
DVIGFX16 display(DVI_RES_320x240p60, adafruit_feather_dvi_cfg);

// Output of pioasm ----

#define fourwire_wrap_target 2
#define fourwire_wrap 5

static const uint16_t fourwire_program_instructions[] = {
    0xa0c3, //  0: mov    isr, null
    0x0005, //  1: jmp    5
            //     .wrap_target
    0x2022, //  2: wait   0 pin, 2
    0x20a2, //  3: wait   1 pin, 2
    0x4002, //  4: in     pins, 2
    0x00c0, //  5: jmp    pin, 0
            //     .wrap
};

static const struct pio_program fourwire_program = {
    .instructions = fourwire_program_instructions,
    .length = 6,
    .origin = -1,
};

static inline pio_sm_config fourwire_program_get_default_config(uint offset) {
    pio_sm_config c = pio_get_default_sm_config();
    sm_config_set_wrap(&c, offset + fourwire_wrap_target, offset + fourwire_wrap);
    return c;
}

// end pioasm output ----

PIO pio = pio1; // libdvi uses pio0 (but has 1 avail state machine if you want to
use it)
uint sm;
uint16_t *framebuf = display.getBuffer();
uint8_t decode[256];
#define DECODE(w) (decode[(w & 0x55) | ((w >> 7) & 0xaa)])

#define BIT_DEPOSIT(b, i) ((b) ? (1<<(i)) : 0)
#define BIT_EXTRACT(b, i) (((b) >> (i)) & 1)
#define BIT_MOVE(b, src, dest) BIT_DEPOSIT(BIT_EXTRACT(b, src), dest)
#define ENCODED_COMMAND(x) ( \
    (BIT_MOVE(x, 0,  0)) | \
    (BIT_MOVE(x, 1,  2)) | \
    (BIT_MOVE(x, 2,  4)) | \
    (BIT_MOVE(x, 3,  6)) | \
    (BIT_MOVE(x, 4,  8)) | \
    (BIT_MOVE(x, 5, 10)) | \
    (BIT_MOVE(x, 6, 12)) | \
    (BIT_MOVE(x, 7, 14)) \
)

#define COMMAND_NOP     (0x00)
#define COMMAND_SWRESET (0x01)
#define COMMAND_CASET   (0x2a)
#define COMMAND_PASET   (0x2b)
#define COMMAND_RAMWR   (0x2c)
#define COMMAND_MADCTL  (0x36)
```

```
#define MADCTL_MY 0x80
#define MADCTL_MX 0x40
#define MADCTL_MV 0x20
#define MADCTL_ML 0x10

void setup() {
  Serial.begin(115200);
  //while(!Serial);
  if (!display.begin()) { // Blink LED if insufficient RAM
    pinMode(LED_BUILTIN, OUTPUT);
    for (;;) digitalWrite(LED_BUILTIN, (millis() / 500) & 1);
  }

  for(int i=0; i<256; i++) {
    int j = (BIT_MOVE(i, 0, 0)) |
            (BIT_MOVE(i, 2, 1)) |
            (BIT_MOVE(i, 4, 2)) |
            (BIT_MOVE(i, 6, 3)) |
            (BIT_MOVE(i, 1, 4)) |
            (BIT_MOVE(i, 3, 5)) |
            (BIT_MOVE(i, 5, 6)) |
            (BIT_MOVE(i, 7, 7));
    decode[i] = j;
  }

  uint offset = pio_add_program(pio, &fourwire_program);
  sm = pio_claim_unused_sm(pio, true);

  pio_sm_config c = fourwire_program_get_default_config(offset);

  // Set the IN base pin to the provided PIN_DATA parameter. This is the data
  // pin, and the next-numbered GPIO is used as the clock pin.
  sm_config_set_in_pins(&c, PIN_DATA);
  sm_config_set_jmp_pin(&c, PIN_CS);
  // Set the pin directions to input at the PIO
  pio_sm_set_consecutive_pindirs(pio, sm, PIN_DATA, 3, false);
  pio_sm_set_consecutive_pindirs(pio, sm, PIN_CS, 1, false);
  // Connect GPIOs to PIO block, set pulls
  for (uint8_t i=0; i<3; i++) {
    pio_gpio_init(pio, PIN_DATA + i);
    gpio_set_pulls(PIN_DATA + i, true, false);
  }
  pio_gpio_init(pio, PIN_CS);
  gpio_set_pulls(PIN_CS, true, false);

  // Shifting to left matches the customary MSB-first ordering of SPI.
  sm_config_set_in_shift(
      &c,
      false, // Shift-to-right = false (i.e. shift to left)
      true,  // Autopush enabled
      16     // Autopush threshold
  );

  // We only receive, so disable the TX FIFO to make the RX FIFO deeper.
  sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);

  // Load our configuration, and start the program from the beginning
  pio_sm_init(pio, sm, offset, &c);
  pio_sm_set_enabled(pio, sm, true);

  // State machine should handle malformed requests somewhat,
  // e.g. RAMWR writes that wrap around or drop mid-data.

  uint8_t cmd = COMMAND_NOP;                  // Last received command
  uint16_t X0 = 0, X1 = display.width() - 1;  // Address window X
  uint16_t Y0 = 0, Y1 = display.height() - 1; // Address window Y
  uint16_t x = 0, y = 0;                      // Current pixel pos.
  union { // Data receive buffer sufficient for implemented commands
    uint8_t  b[4];
```

```
    uint16_t w[2];
    uint32_t l;
  } buf;
  int8_t bufidx = -1; // Current pos. in buf.b[] array (or -1 = full)

  for (;;) {
    uint16_t ww = pio_sm_get_blocking(pio, sm); // Read next word (data & DC
interleaved)
    if ((ww & 0x2)) { // DC bit is set, that means it's data (most common case,
hence 1st)
      if (bufidx >= 0) { // Decode & process only if recv buffer isn't full
        buf.b[bufidx] = DECODE(ww);
        // Buffer is filled in reverse so byte swaps aren't needed on uint16_t
values
        if (--bufidx < 0) { // Receive threshold reached?
          switch (cmd) {
           case COMMAND_CASET:
            // Clipping is not performed here because framebuffer
            // may be a different size than implied SPI device.
            // That occurs in the RAMWR condition later.
            X0 = buf.w[1]; // [sic.] 1 because buffer is loaded in reverse
            X1 = buf.w[0];
            if (X0 > X1) {
              uint16_t tmp = X0;
              X0 = X1;
              X1 = tmp;
            }
            break;
           case COMMAND_PASET:
            Y0 = buf.w[1]; // [sic.] 1 because buffer is loaded in reverse
            Y1 = buf.w[0];
            if (Y0 > Y1) {
              uint16_t tmp = Y0;
              Y0 = Y1;
              Y1 = tmp;
            }
            break;
           case COMMAND_RAMWR:
            // Write pixel to screen, increment X/Y, wrap around as needed.
            // drawPixel() is used as it handles both clipping & rotation,
            // saves a lot of bother here. However, this only handles rotation,
            // NOT full MADCTL mapping, but the latter is super rare, I think
            // it's only used in some eye code to mirror one of two screens.
            // If it's required, then rotation, mirroring and clipping will
            // all need to be handled in this code...but, can write direct to
            // framebuffer then, might save some cycles.
            display.drawPixel(x, y, buf.w[0]);
            if (++x > X1) {
              x = X0;
              if (++y > Y1) {
                y = Y0;
              }
            }
            bufidx = 1; // Reset buffer counter for next pixel
            // Buflen is left as-is, so more pixels can be processed
            break;
           case COMMAND_MADCTL:
            switch (buf.b[0] & 0xF0) {
             case MADCTL_MX | MADCTL_MV:             // ST77XX
             case MADCTL_MX | MADCTL_MY | MADCTL_MV: // ILI9341
              display.setRotation(0);
              break;
             case MADCTL_MX | MADCTL_MY: // ST77XX
             case MADCTL_MX:             // ILI9341
              display.setRotation(1);
              break;
             case MADCTL_MY | MADCTL_MV: // ST77XX
             case MADCTL_MV:             // ILI9341
              display.setRotation(2);
```

```
              break;
             case 0:         // ST77XX
             case MADCTL_MY: // ILI9341
              display.setRotation(3);
              break;
            }
            break;
          }
        }
      }
    } else { // Is command
      cmd = DECODE(ww);
      switch (cmd) {
       case COMMAND_SWRESET:
        display.setRotation(0);
        x = y = X0 = Y0 = 0;
        X1 = display.width() - 1;
        Y1 = display.height() - 1;
        break;
       case COMMAND_CASET:
        bufidx = 3; // Expecting two 16-bit values (X0, X1)
        break;
       case COMMAND_PASET:
        bufidx = 3; // Expecting two 16-bit values (Y0, Y1)
        break;
       case COMMAND_RAMWR:
        bufidx = 1; // Expecting one 16-bit value (or more)
        x = X0;     // Start at UL of address window
        y = Y0;
        break;
       case COMMAND_MADCTL:
        bufidx = 0; // Expecting one 8-bit value
        break;
       default:
        // Unknown or unimplemented command, discard any data that follows
        bufidx = -1;
      }
    }
  }
}

void loop() {
}
```

```
.program fourwire
; Sample bits using an external clock, and push groups of bits into the RX FIFO.
; - IN pin 0 is the data pin   (GPIO18)
; - IN pin 1 is the dc pin     (GPIO19)
; - IN pin 2 is the clock pin  (GPIO20)
; - JMP pin is the chip select (GPIO21)
; - Autopush is enabled, threshold 8
;
; This program waits for chip select to be asserted (low) before it begins
; clocking in data. Whilst chip select is low, data is clocked continuously. If
; chip select is deasserted part way through a data byte, the partial data is
; discarded. This makes use of the fact a mov to isr clears the input shift
; counter.
flush:
    mov isr, null         ; Clear ISR and input shift counter
    jmp check_chip_select ; Poll chip select again
.wrap_target
do_bit:
    wait 0 pin 2          ; Detect rising edge and sample input data
    wait 1 pin 2          ; (autopush takes care of moving each complete
    in pins, 2            ; data word to the FIFO)
check_chip_select:
```

```
    jmp pin, flush        ; Bail out if we see chip select high
.wrap
```

Screensavers

In the olde days™ of CRT monitors, good computer hygiene meant installing a screensaver to prevent image burn-in. Early operating systems didn't even provide this...such programs had to be purchased! We've replicated the nostalgic essence of a few popular classics...only this time, the code's all free.

With the PicoDVI library installed as previously described (https://adafru.it/18AB), you'll find these in Arduino's Examples menu...

File→Examples→PicoDVI - Adafruit Fork→screensavers

If you'd prefer to just mess around with the demos, we also have pre-compiled UF2 files for the **Feather RP2040 DVI** boards at 400x240 resolution and for the **Pico with the DVI PiCowbell** at 320x240 for each example below.

The examples are all written for the **Feather RP2040 DVI** using 320x240 pixels, but easily adapted for other RP2040 boards, and for 400x240 pixel resolution with a little extra work. See the earlier 8bit_double_buffer example for an explanation.

We won't be showing the code for these examples "inline" here because the embedded graphics data makes them huge...you'd be scrolling for weeks. There are comments in the code and some additional notes at the end of this page.

Flying Toasters

The classic-est of the classics! This is what made folks want to buy a screensaver.

Not emulated, just an approximation of the original.

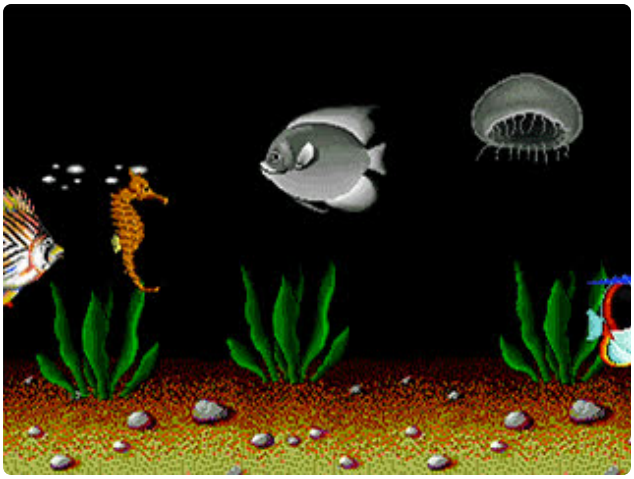
toasters.uf2

https://adafru.it/18AD

toasters_dvibell.ino.uf2

https://adafru.it/19JE

Aquarium



The other classic. Not as zany™ as the toasters, but soothing to watch.

aquarium.uf2

https://adafru.it/18AE

aquarium_dvibell.ino.uf2

https://adafru.it/19JF

TV Host

Also not a classic PC screensaver but it had to be done.

Inspired by the trash-heap screens in Max Headroom: 20 Minutes into the Future (1985). Been noticing more working flatscreen TVs and monitors left out by the curb lately. Free pixels! What could we make of them? This was one such idea.

tv-host.uf2

https://adafru.it/18AF

tvhost_DVIBELL.ino.uf2

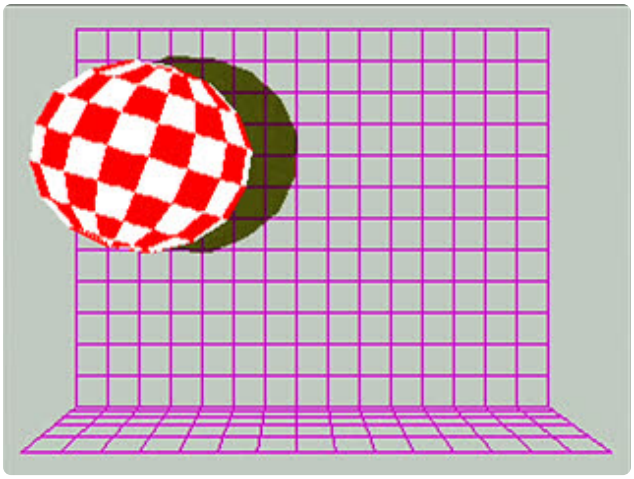
https://adafru.it/19Ka

Here's a version that's just the rotating cube background:

cube-background.uf2

https://adafru.it/18AG

Boing!



The Amiga demo that started it all.

(This UF2 file is compiled at 320x240 for aspect ratio authenticity.)

boing.uf2

https://adafru.it/18AH

boing_dvibell.ino.uf2

https://adafru.it/19Kb

Logo Bounce

Okay, it's not a classic PC screensaver, but had to include this as the code's the simplest of the lot, and also it shows crisp high-resolution (640x480 or 800x480) monochrome animation. This example is too high a resolution to run on the Pico.

Hi-res monochrome would be ideal for making retro vector-style games like Asteroids or Battlezone.

Here's a ready-made UF2 file for Feather RP2040 DVI:

logobounce.uf2

https://adafru.it/18AC

Notes

All of these examples do something similar: graphics “sprites” are embedded as tables in the code, and then bitmap-drawing functions in Adafruit GFX “blit” these to the screen. Double-buffered animation prevents flicker; everything appears buttery smooth.

The “TV Host” example is distinct in that it also lays down some vector-style graphics first, then overlays large sprites atop this. The backdrop is doing some actual 3D perspective but only in the barest sense.

None of these examples is especially “crafty” — the approach is brute-force, using large sprite tables as a tradeoff to keep the code simpler to read. The demos are nonetheless impressive for a modestly-priced microcontroller board. If slapdash code can do this, then advanced programmers will likely be creating some stunning work!

The sprite tables were generated using a mix of ImageMagick and/or Python. We don’t have a well-written universal tool for this because sometimes that’s just the nature of Python…quickly writing a sloppy one-off program for a special task.

ImageMagick was mostly used for the 1-bit sprite masks, for example:

```
magick mask.png -define h:format=gray -depth 1 mask.h
```

The output needs a little editing (see the sprite.h/sprites.h header file that’s a part of each example), but ImageMagick does most of the work.

Here’s a little Python code (not CircuitPython, but “desktop” Python) that extracts the color palette from an 8-bit PNG or GIF and converts it to the “RGB565” color format used by PicoDVI and Adafruit GFX, then follows up with a table of sprite pixel values:

```
from PIL import Image, ImagePalette

with Image.open("dragon.png") as img:

    pal = img.getpalette()
    print("const uint16_t palette[] = {")
    for i in range(len(pal) // 3):
        r = pal[i * 3 + 0] &gt;&gt; 3
        g = pal[i * 3 + 1] &gt;&gt; 2
        b = pal[i * 3 + 2] &gt;&gt; 3
        rgb = (r &lt;&lt; 11) | (g &lt;&lt; 5) | b;
        print("0x%04X, " % (rgb), end="")
    print("};\n")

    print("const uint8_t sprite[] = {")
    for y in range(img.size[1]):
        for x in range(img.size[0]):
```

```
            p = img.getpixel((x, y))
            print("0x%02X, " % (p), end="")
    print("};\n")
```

The output of this code is then redirected to a file, such as:

```
python convert.py &gt; sprite.h
```

That file's contents will be a sloppy mess! To keep this simple, rather than formatting nicely in the Python code, the output is run through the clang-format tool to clean it up:

```
clang-format -i sprite.h
```

The mask data (for programs that need it) was then appended.

Sometimes ImageMagick can be used for the sprite conversion as well, but it tends to be headstrong about optimizing and reordering colors. There's probably settings to disable those behaviors, but doing the conversion in Python was simple enough.

The boing example is a little different because the code was adapted from an earlier project for PyPortal. That one uses "packed" data, two pixels per byte.

---

file_access

Yeehaw!

Notice how all the screensaver examples embed their graphics data as tables within the code. Any change in graphics requires new tables. What if we could load graphics…or any other data our program might need…from files instead?

Installing **CircuitPython** initializes a board's flash storage like a small USB drive. We then have **Arduino** libraries that can read and write files in this space.

Combining this with PicoDVI is possible, but is experimental and failure-prone. Either one of these tasks — DVI video or USB storage — is pretty demanding, and combining the two is asking a lot. In no case will it damage the board, but at worst there's a small chance of wiping out the flash storage and having to re-load everything. Understand **this is wild west stuff.** We'll walk through some best practices to make this less perilous…

First, you'll need up-to-date versions of several packages:

Tools→Board→Boards Manager...

- The RP2040 Arduino "core" must be version 3.3.0 or later. Setup and installation is explained in the Feather RP2040 DVI guide (https://adafru.it/18AJ).

Sketch→Include Library→Library Manager...

- PicoDVI library 1.1.0 or later.
- Adafruit_SPIFlash 4.2.0 or later.
- Adafruit_CPFS 1.1.0 or later.

Then head to the CircuitPython downloads page (https://adafru.it/Em8) and grab the current version for your board. Keep this file around. As mentioned above, there's a small chance of things going awry and needing to re-initialize a board's flash storage.

Prepare Hardware

Install CircuitPython as you world normally. On most RP2040 boards, this involves holding down the BOOT button while connecting USB or tapping the reset button. A drive called RPI-RP2 will appear on your computer. Drag the CircuitPython .UF2 file to this drive and let it work. After several seconds, RPI-RP2 is replaced with a CIRCUITPY drive. Success! Now there's a viable filesystem to hold files.

Drag files for your project to the CIRCUITPY drive and organize to your liking. It's prudent to **keep a clean copy of everything organized on your computer, should a reinstall be necessary.**

The file_access example doesn't require any additional files. All this example does is print a directory listing. But even a fresh CircuitPython install will have a few items present.

Compile and Upload Sketch

Open the file_access sketch in the Arduino IDE. Select your board type from the "Tools" menu rollover...

Tools→Board→Raspberry Pi RP2040 Boards→(board name)

Then...important detail or the sketch won't compile...select the TinyUSB stack, which helps make the CIRCUITPY drive accessible to Arduino code and the host computer...

Tools→USB Stack→Adafruit TinyUSB

Then you can compile and upload to the board as you would any other sketch.

In Action

To keep the code brief and instructive, all this example does is print a listing of the CIRCUITPY root directory to a connected DVI display. But...selecting or previewing different files from the attached host computer, you'll see it prints the listing again. Selecting a file updates the last "touch" time for that file, which constitutes a change. You can also try dragging files to or removing files from the CIRCUITPY drive, and should get an updated listing with each change.

This example uses a text mode, but the same principles can be applied to graphical programs as well. Most won't even need the change-detect behavior...whatever data is loaded on start-up, that's what you get in that case.

Code Explainer

The PicoDVI parts are already explained in other examples and won't be covered here. Only the file-specific additions are mentioned.

First up, an extra library header and global object are declared:

```
#include &lt;Adafruit_CPFS.h&gt; // For accessing the CIRCUITPY drive

FatVolume *fs = NULL; // CIRCUITPY flash filesystem, as a FAT pointer
```

Adafruit_CPFS (**C**ircuit**P**ython **F**ile **S**ystem) is an Arduino library that helps make the CIRCUITPY flash filesystem available both to our Arduino sketch and presents it to a host computer over USB. `fs` in this case is a global pointer to that filesystem object; initially NULL as we've not set it up yet.

A little later, in the `setup()` function, `fs` is then initialized by calling `Adafruit_CPFS::` `begin` like so:

```
fs = Adafruit_CPFS::begin(true, -1, NULL, false);
```

This is an unusual syntax in that all Adafruit_CPFS functions are "static" — they do not require declaring a special object type (though one can if desired), the flipside being that all function calls to that library begin with `Adafruit_CPFS::`, with the two colons.

More of interest here are all the arguments passed to the `begin()` function:

- The initial `true` argument tells the library to make the filesystem available both to our sketch and to a host computer over USB.
- The `-1` and `NULL` arguments have no meaning here. The Adafruit_CPFS library works on a variety of different hardware, and these are specific to other devices. Necessary but ignored here.
- The trailing `false` argument is critical when combining Adafruit_CPFS and PicoDVI on an RP2040 microcontroller. This keeps the second core running at all times (where the DVI-generating code resides), where CPFS would normally want to pause it during writes.

If this were a non-PicoDVI project, all those arguments would be unnecessary, and `begin()` would just be called like so:

```
fs = Adafruit_CPFS::begin();
```

A few lines down, after the DVI display has been initialized, the value of the `fs` variable is checked to see whether the flash filesystem is present and working:

```
if (fs == NULL) { // If CIRCUITPY filesystem is missing or malformed...
    // Show error message &amp;amp; fast blink LED to indicate problem. Full stop.
    display.println("Can't access board's CIRCUITPY drive.");
    display.println("Has CircuitPython been previously installed?");
    for (;;) digitalWrite(LED_BUILTIN, (millis() / 250) &amp;amp; 1);
  } // else valid CIRCUITPY drive, proceed...
```

And, after that point, `fs` can be treated like a normal filesystem object, say if you've used the Arduino SD library before. Only difference is, because it's a pointer to an object, one uses C's indirection syntax `->` rather than `.`:

```
fs-&amp;gt;open("settings.json", FILE_READ);
```

Avoiding The Crash

The prospect of sometimes re-loading a board is frustrating but also interesting how it came about. The RP2040 chip has no internal flash memory; code is read from external flash, which also holds the CIRCUITPY flash filesystem. It's not possible to simultaneously write or erase one part of flash while executing code from another… normally, all activity has to stop for this, which PicoDVI (generating time-critical signals) abhors. Getting all these to even sort of play nice together took some work.

Here are some steps one can take to avoid The Crash:

- Consider having your Arduino sketch treat the CIRCUITPY drive as **read-only**; steer clear of writing or creating new files, and open files in `FILE_READ` mode only. There's nothing preventing writing files, and you're welcome to try and see how it goes, but you take your chances. This is why the CP/M emulator project uses a microSD card (https://adafru.it/18AK) rather than the internal flash; there's no contention for resources that way.

- For a crucial situation or live demo, don't power the board off a computer's USB port, and instead use a passive source like a USB power supply or a LiPoly battery. If there's nothing trying to mount the drive and touch files, operation should be reliable.

- Consider adding a **switch or button** to your circuit that can be **polled on startup**, and only present the filesystem over USB in one state or the other using the first argument to `Adafruit_CPFS::begin()` (see code snippet above). An initial `true` value makes the drive computer-writable, while `false` makes it so only the Arduino code sees the drive.

- Perhaps take the switch idea one step further and don't even start the PicoDVI display when the drive is mounted. With the switch one way, the computer sees the CIRCUITPY drive and you can load up new files, but there's no display. Switched the other way, video works but the computer won't see the drive. This should be fairly robust.