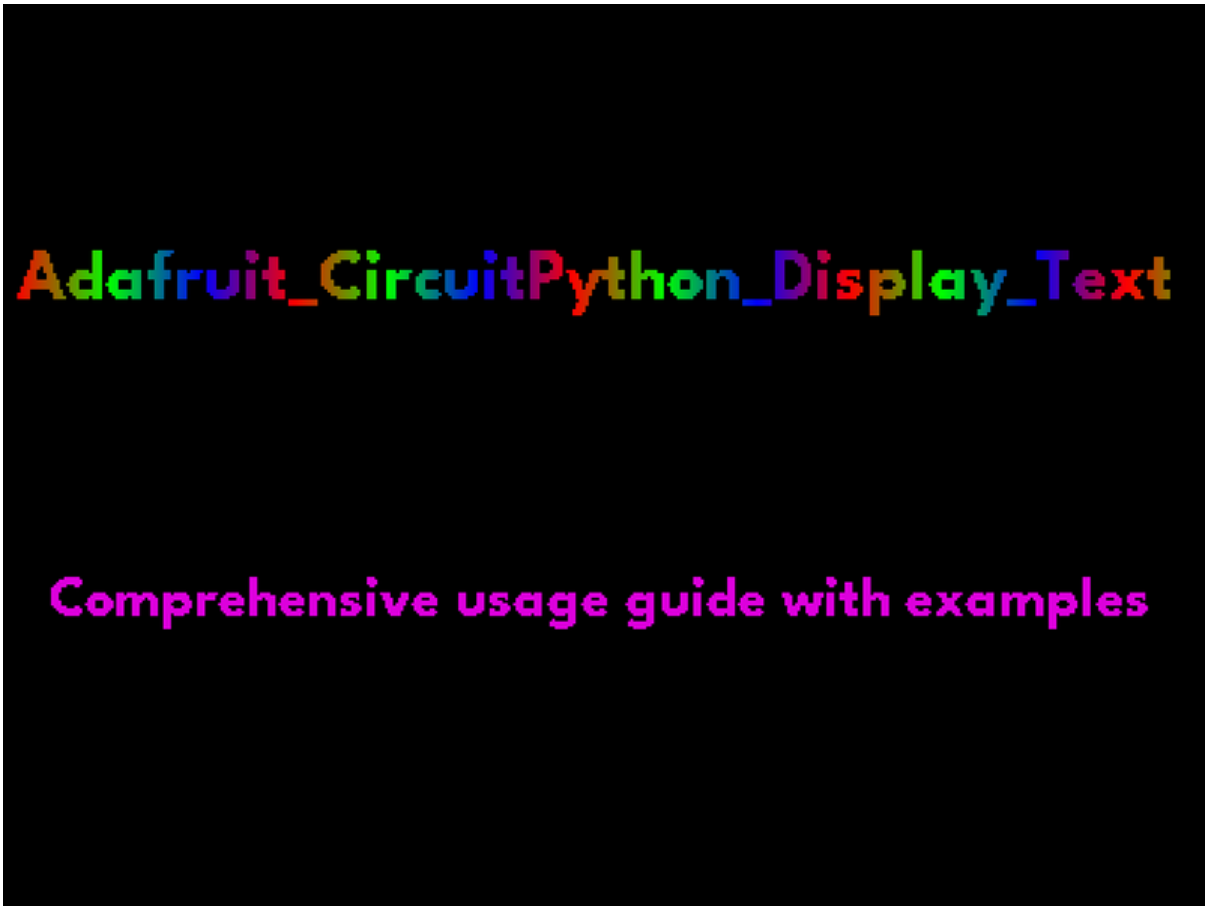
CircuitPython Display_Text Library

Created by Tim C

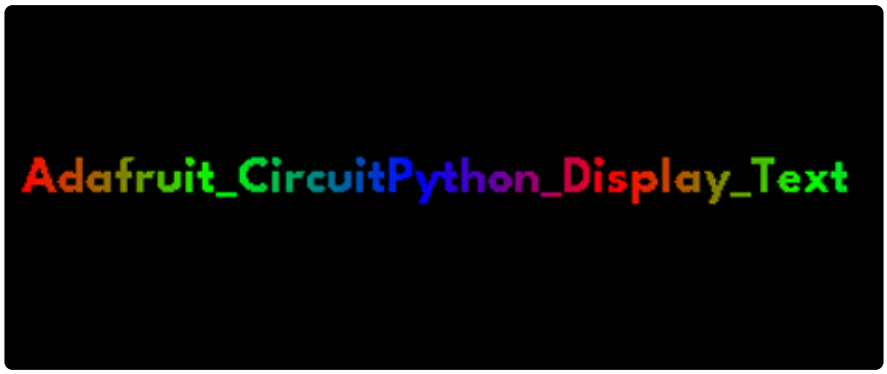


https://learn.adafruit.com/circuitpython-display_text-library

Last updated on 2024-06-03 03:20:42 PM EDT

Table of Contents

Overview



The Adafruit_CircuitPython_Display_Text (https://adafru.it/FiA) library allows you to easily add text to your `displayio` based CircuitPython projects. If you have no prior experience with `displayio`, you should check out the Displayio Basics Guide (https://adafru.it/EGh) first to learn the basic concepts.

This guide will show you how to install the library and go over all of the features and what they do.

Other Helpful Resources

- Display_Text docs page (https://adafru.it/R9d)
- Repository Example scripts (https://adafru.it/R9e)
- Example scripts from this guide (https://adafru.it/R9f)
- Bitmap_Font library (https://adafru.it/Fiv)

Parts

Here are some displays that work well with CircuitPython and the displayio library:

Adafruit PyPortal - CircuitPython Powered Internet Display

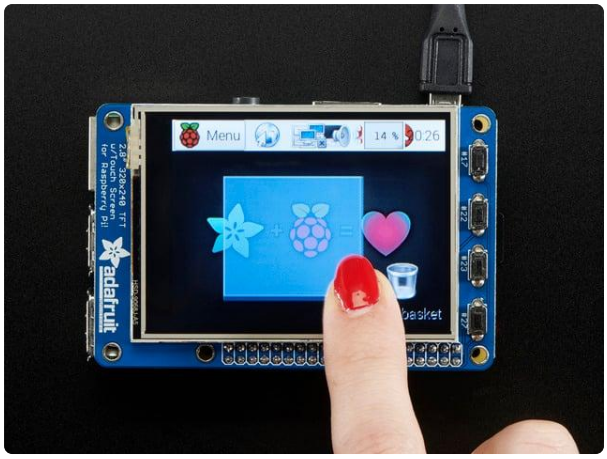
PyPortal, our easy-to-use IoT device that allows you to create all the things for the “Internet of Things” in minutes. Make custom touch screen interface...

https://www.adafruit.com/product/4116

Adafruit MagTag - 2.9" Grayscale E-Ink WiFi Display

The Adafruit MagTag combines the new ESP32-S2 wireless module and a 2.9" grayscale E-Ink display to make a low-power IoT display that can show data on its screen even when power...

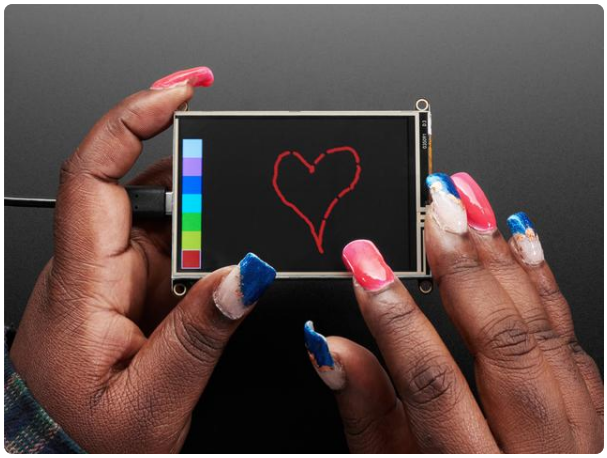
https://www.adafruit.com/product/4800



PiTFT Plus Assembled 320x240 2.8" TFT + Resistive Touchscreen

Is this not the cutest little display for the Raspberry Pi? It features a 2.8" display with 320x240 16-bit color pixels and a resistive touch overlay. The plate uses the high...

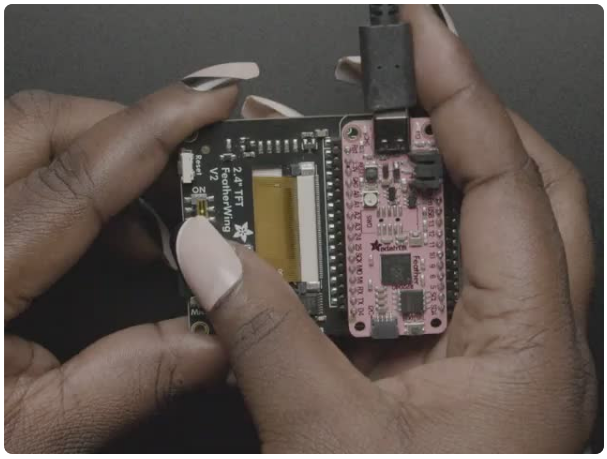
https://www.adafruit.com/product/2298



Adafruit TFT FeatherWing - 3.5" 480x320 Touchscreen for Feathers

Spice up your Feather project with a beautiful 3.5" touchscreen display shield with built in microSD card socket. This TFT display is 3.5" diagonal with a bright 6 white-LED...

https://www.adafruit.com/product/3651



TFT FeatherWing - 2.4" 320x240 Touchscreen For All Feathers

A Feather board without ambition is a Feather board without FeatherWings! Spice up your Feather project with a beautiful 2.4" touchscreen display shield with built in microSD card...

https://www.adafruit.com/product/3315

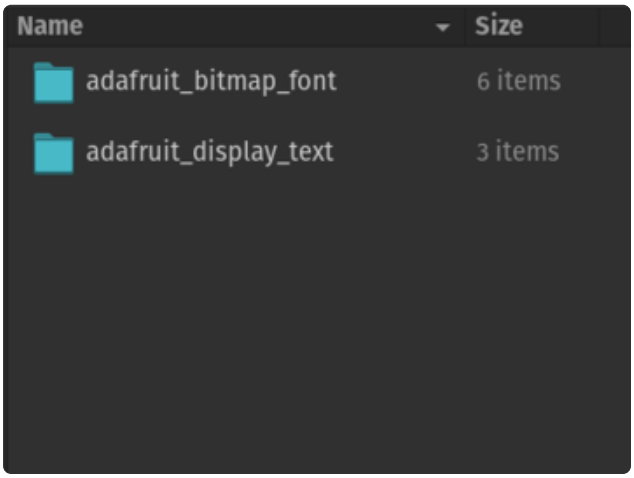
Setup

You should install the latest version of CircuitPython for your device as well as the newest versions of the libraries which will be used.

CircuitPython

The topics covered in this guide apply to many different devices. Follow the instructions here (https://adafru.it/Amd) to get the newest version of for your device. Find the latest version for your device at circuitpython.org/downloads (https://adafru.it/Em8).

Libraries



Next you'll need to install the required libraries. Download the latest library bundle (https://adafru.it/ENC) and then copy them into the **lib** folder on your device. This guide will be focusing on these two:

adafruit_display_text
adafruit_bitmap_font
You may have other libraries present on your device, but you must have at least these two.

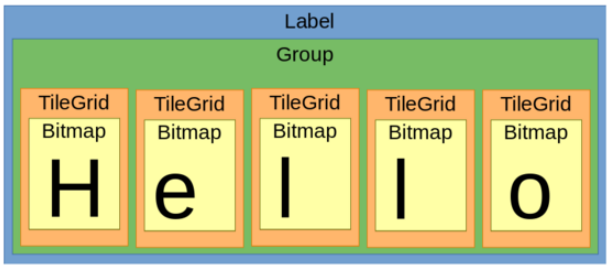
Types of Labels

The adafruit_display_text library currently has two different types of label functions in it. This guide goes over those label functions and the differences between them. If you are less concerned about the details and just want to know which to use for your new project, go with BitmapLabel (https://adafru.it/R9A).

Label

Starting from CircuitPython 7.0.0 there is no longer a max_size restriction for Group objects.

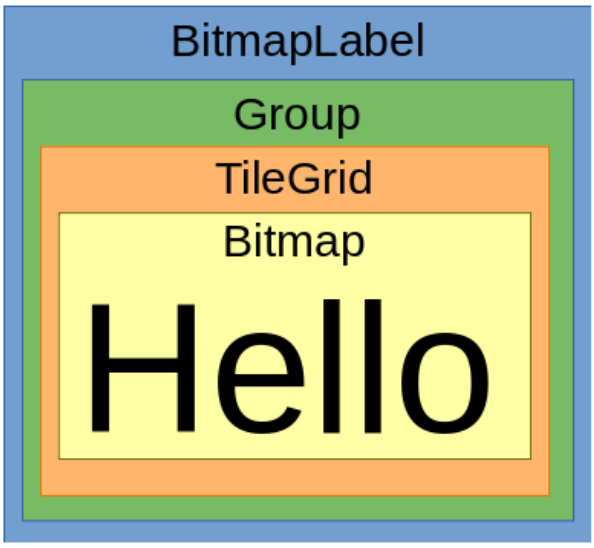
This is the original Label class. Each glyph within the text is stored in its own Bitmap and TileGrid objects, and those all get put into a single Group object. In prior versions of CircuitPython there was a parameter `max_glyphs` that enforced a limitation in the underlying Group that limited the number of items in the Group. In CircuitPython 7.0.0 this limitation has been removed from Group and therefore also no longer applies to Label.



If you set a `background_color`, then the background will get its own TileGrid and Bitmap as well. The diagram above depicts a Label with no background Bitmap in it.

BitmapLabel

This is a newer class that was introduced more recently after many features were added into the original Label class. In the BitmapLabel, all of the glyphs are stored inside of a single Bitmap and TileGrid. This tends to result in lower memory usage, especially for long strings.



Starting from CircuitPython 7.0.0 there is no longer a max_size restriction for Group objects.

BitmapLabel typically will use a little bit less RAM -- it can be helpful sometimes in larger projects to switch from Label to BitmapLabel to save more RAM for your projects other needs.

OutlinedLabel

Starting with the release version **3.1.0** the Adafruit_Display_Text (https://adafru.it/19NC) Library supports an `OutlinedLabel` which lets you make text with an outline stroke of a specified size and color.

Here is a screenshot of an `OutlinedLabel` with pink text `color` and green `outline_color`. The code below is an example of this.

```
text_area = OutlinedLabel(
    terminalio.FONT,
    text="Outlined\nLabel",
    color=0xFF00FF,
    outline_color=0x00FF00,
    outline_size=1,
    scale=5,
)
```

Scrolling Label

Starting with the release version **2.22.0** the Adafruit_Display_Text (https://adafru.it/19ND) library supports `ScrollingLabel`. The `ScrollingLabel` class is a label that can have a specified maximum number of characters showing at a time and will scroll through a larger message, like a marquee.

The ScrollingLabel depicted above is initialized with the following code:

```
text = "Hello world CircuitPython scrolling label"
my_scrolling_label = ScrollingLabel(
    terminalio.FONT,
    text=text,
    max_characters=20,
    animate_time=0.1,
    scale=2
)
while True:
    my_scrolling_label.update()
```

Advanced Color Masking

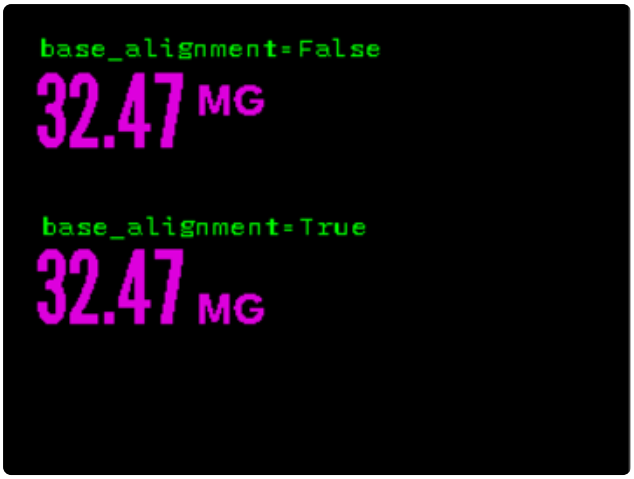
BitmapLabel can be used with `None` value for `color` and any opaque value for `background_color` to produce a transparent "cutout" of the text in the label. You can put rainbows or other interesting things in the background, and layer the BitmapLabel on top to produce fancy text graphics. See the example here (https://adafru.it/Xe7).

Label Placement

There are a few different ways to place your text with display_text labels. All of these settings are available for both Label and BitmapLabel.

x, y coordinates

The label can be placed using x, y coordinates. The default origin is located on the far left horizontally, and half the height vertically as shown in the image above.



Baseline Alignment

The `base_alignment` parameter can be set to `True` to change the vertical origin to point to the baseline of the text. This is helpful for lining up fonts of different sizes nicely next to each other. The default value is `False`.

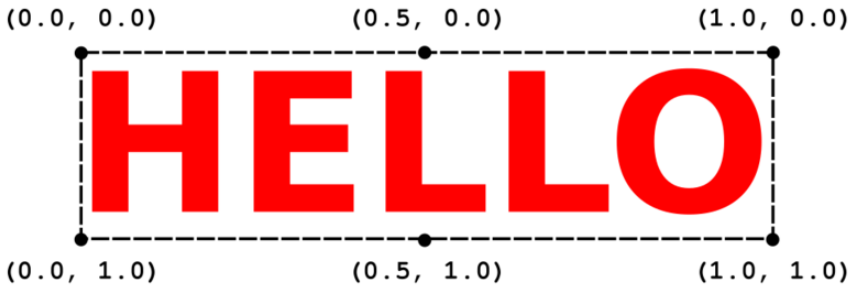
Descenders

Many fonts have glyphs with descenders that drop down below the baseline. This image illustrates some of them:

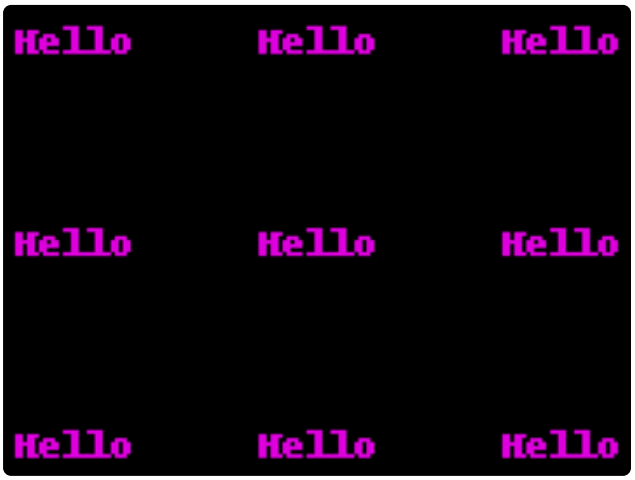
Not all fonts use all of the same descenders on their glyphs. So your font may look different, but the concept should work the same for any that it does have. This example shows 'y' glyphs descending below the baseline and the effect of the `base_alignment` parameter on them.

Anchored Positioning

If you'd like more precise control over the placement of your text, you can use `anchor_point` and `anchored_position` instead of `x` and `y` coordinates. With this method, you can set the origin point anywhere you want within your Label. It's particularly useful for centering text within the display, or aligning it to the right edge of the display.



`anchor_point` is a tuple containing two float values. This is the point that your Label will get positioned relative to. `0.0` being left and top, and `1.0` being right and bottom with values between representing their location within the rectangle that makes up your label. The default is `(0.0, 0.0)` which is the top left corner.



Anchored Position

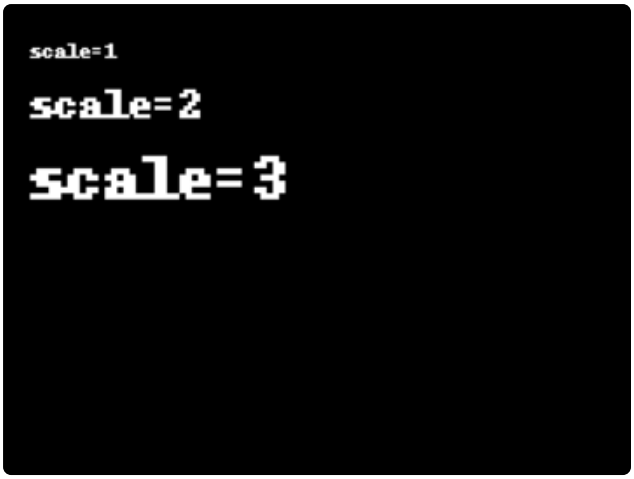
When you set the `anchored_position`, the label will move so that its `anchor_point` is aligned on top of the coordinates that you set for the `anchored_position`. The value is a tuple containing two integers that represent pixel coordinates on the display. There is a similar example in the library repository (https://adafru.it/PEa).

Label Appearance

There are many parameters and properties you can use to control the way your text looks. This page will list them and show examples of each. All of these settings may be used on both BitmapLabel and Label.

Font

The `font` parameter is used to set the typeface for your text. The default value is `terminalio.FONT`, which is the built-in font for the system. Different builds of CircuitPython can have different built-in fonts, though many of them have the same one. Blinka_Displayio has a built-in font as well. You can load custom font files from pcf and bdf font files using the adafruit_circuitpython_bitmap_font library.



Scale

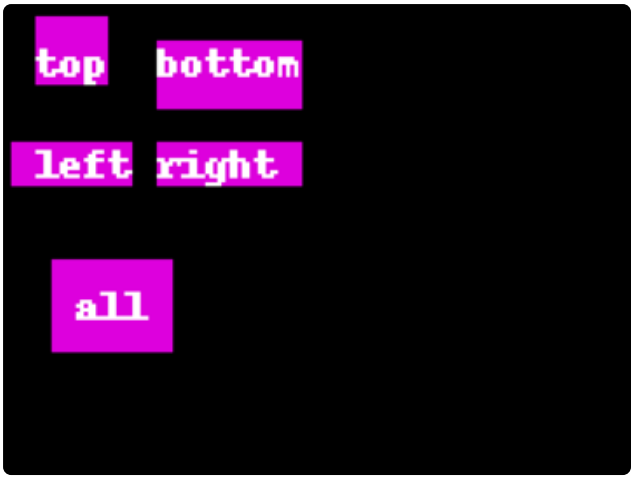
The `scale` parameter is inherited from Group. It can be used to make the label bigger. The default value is `1`. Set it to a larger integer to make the text bigger by that scale factor.

Color

`color` is an optional parameter and property that controls the color of your text. The value should be a hex color value, and the default is `0xFFFFFF` or white.

Background Color

The optional parameter and property `background_color` will cause a background box to be drawn containing your text. The default is `None`, which causes no background to be drawn. Set it to a color hex value to enable, i.e. `0xFF00FF`.

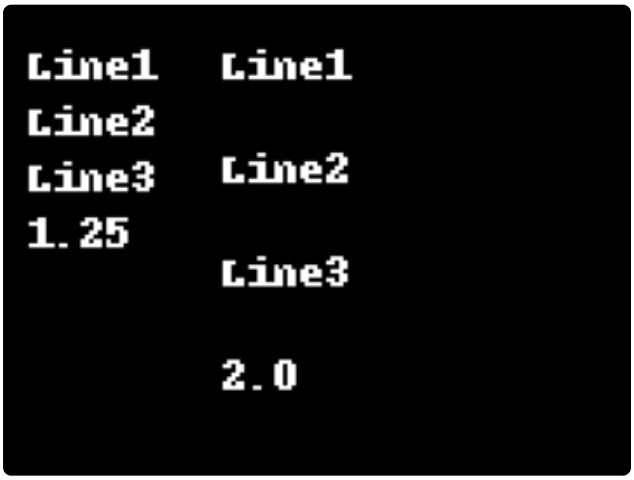


Padding

There are 4 parameters that can be used to adding padding to the insides of your label: `padding_top`, `padding_bottom`, `padding_left`, and `padding_right`. You can use any single one, or any combination of them.

Background Tight

The `background_tight` parameter specifies that you want the background box to be drawn as small or tight as possible. When this is set to `True`, the padding values will be ignored. Also, if there are no glyphs used with descender sections, then the background won't include space for them. The default value is `False`, which will include space for the descender even if your text does not have any of them.



Line Spacing

The optional `line_spacing` parameter and matching property are used to specify how much space should appear between different lines of text. You can use `"\n"` in your text to make a new line. The default value is `1.25`, which looks pretty good with the built-in font. You can go smaller to pack the lines closer, or bigger to add some extra blank room between lines.

Advanced Font Customization

Custom fonts are achieved with the Adafruit_Circuitpython_Bitmap_Font (https://adafru.it/Fiv) library. There are two types of custom font files supported: `pcf` and

`bdf`. You can find font files online and use converters to get them to the correct formats for use with display text labels. This guide covers the process:

Custom Fonts for CircuitPython Display

https://adafru.it/E7E

Overview

More Fonts

Are you looking to display new fonts on your PyPortal? You can use just about any font on your computer or downloaded from the internet. This guide will walk you

through generating bitmap fonts using the FontForge open-source (https://adafru.it/DZk) project.

Why Bitmaps?

PyPortal uses the CircuitPython Bitmap Font Library (https://adafru.it/DZl) to render "live" text on the display. A bitmap font stores each character as an array of pixels. Bitmap fonts are simply groups of images. For each variant of the font, there is a complete set of images, with each set containing an image for each character.

Computers, on the other hand, use variable size 'TrueType' or 'Postscript' fonts, where there's a mathematical algorithm that defines each character, so it can be drawn at any size.

Font Forging

This is where FontForge comes into play. FontForge is an open-source font editor for Windows, Mac OS and GNU+Linux. It features tools for converting existing fonts into different font formats.

Getting Started with FontForge

Head on over to the fontforge page (https://adafru.it/DZm) and download the app for your platform. You can choose to donate, subscribe via email or simply click the "**Subscribe/Confirm and Download**" button (no need to enter an email). Follow along with the detailed installation guide to get setup with FontForge.

Download FontForge

https://adafru.it/DZn

Where Do I Get Fonts?

Here's a list of some neat places to obtain some fresh fonts.

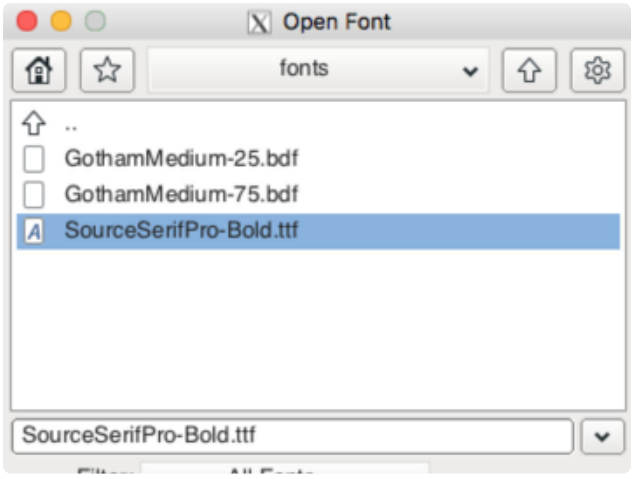
- The Inter typeface family (https://adafru.it/1a0V)

- Font Squirrel (https://adafru.it/DZo)
- Google Fonts (https://adafru.it/DZp)
- Adobe Fonts (https://adafru.it/DZq)
- DaFont (https://adafru.it/DZr)
- Font Library (https://adafru.it/DZs)

Use FontForge

Demo Walkthrough

In this example, we're going to convert a **.TTF** (TrueType Format) into a **.BDF** (Bitmap Distribution Format). I'm using an open licensed font downloaded from Google Font, Source Serif Pro (https://adafru.it/19zc).



Open Font

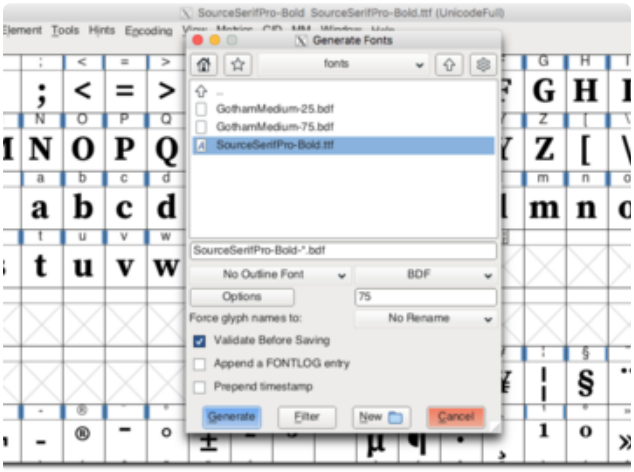
Use the **file** menu and choose **Open Font** from the list. Navigate to a directory where your desired font resides. Select the font and open it.

Set Font Size

From the **element** menu, select **Bitmap Strikes Available**. In this dialog, you will need to specific how large you want your font to be. The font size is fixed with Bitmap fonts, so if you want to use different sizes, you'll need to make separate files.

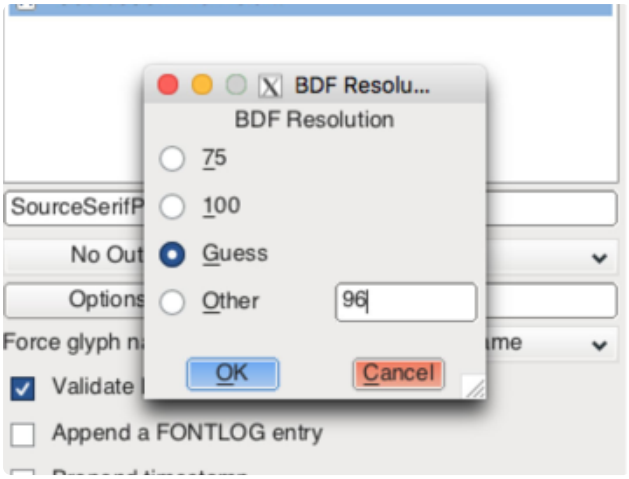
Generate Bits

From the **element** menu, select **Regenerate Bitmap Glyphs**. Similar to the previous dialog, enter the font size of your liking. You can make it smaller here. Be aware, values too small will not generate BDF's.



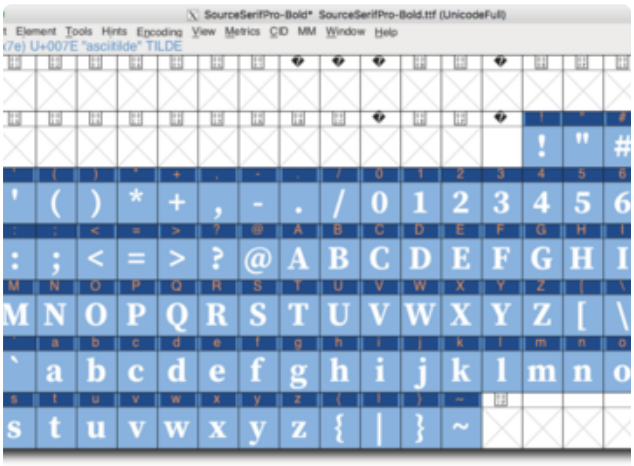
Export Converted Font

From the **file** menu, select **Generate Fonts**. In the dialog, select **No Outline Font** and **BDF** from the dropdown options. Use the navigation UI to save the file in your directory of choice. Click the **generate** button to save the file.



BDF Resolution

This dialog menu will pop up after clicking generate. You can choose one of the options from the list. If you'd like a different font size, you can enter that in the **Other** labeled input box. Click **OK** to save it!



Optimize File Size

If you take a look at the file size of the .bdf, it's roughly around 900K – That can be a bit larger than needed, especially if you plan to store a lot of image and sound assets. In cases where you need to save on every byte, you can optimize the file size of your fonts by selecting only the characters you want to use. If you scroll through the full list of glyphs, you'll see there's extra special characters – A whole bunch of them! If you don't need them in your project, just select "space" (the glyph just before "!") plus the basic set of upper/lower and alphanumeric characters. You can click + hold and drag to make selections easier. With them selected, go through these steps:

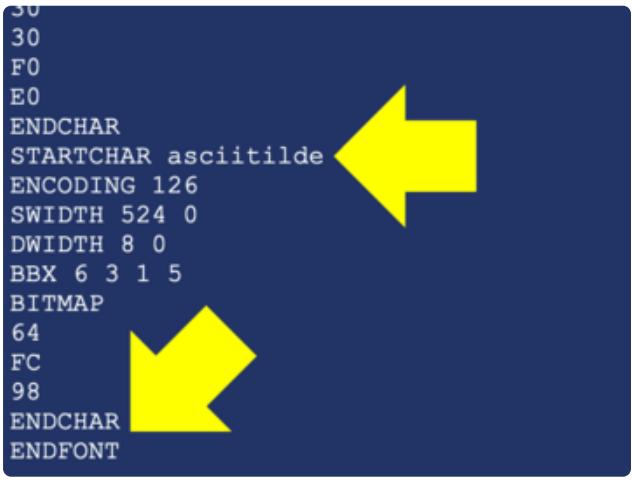
1. Select the glyphs you want to keep
2. Use **Edit→Select→Invert Selection** to change the selection to the unwanted glyphs.
3. Use **Encoding→Detach & Remove Glyphs...** to remove the unwanted glyphs. (You'll have to re-load your original font file to undo this step)
4. Use **Element→Regenerate Bitmap** to reprocess the glyphs.
5. Use **File→Generate Font** to save the reduced version of the file

Make sure your font contains the letter capital M

Make sure your final font contains the letter capital **M**, which is used to estimate the height of letters in the font. Otherwise, the font will be incompatible with **adafruit_display_text** and give an error like `AttributeError: 'NoneType' object has no attribute 'height'`

Optimize File Size (Manually)

If you prefer, you can also use a text editor to remove glyphs from a **.bdf** file. BDF files are just text!



Open a BDF file and search for "**asciitilde**" — this is usually the highest plain-ASCII-value glyph we want to preserve. A few lines down there will be an "**ENDCHAR**" line.

Delete everything after the **ENDCHAR** line, then add a line containing **ENDFONT**. That's it! Save the file, which is usually just a small fraction of the original size.

You won't get any accented characters or special punctuation this way, so it's not always the right thing for every situation. For the majority of plain-text programs though, this can really help stretch your CIRCUITPY drive space!

Font Colors

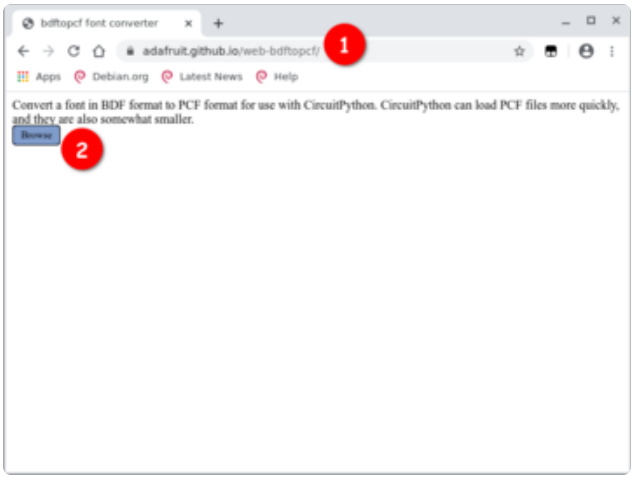
The color of the fonts can be setup in your code. The CircuitPython library uses HEX color codes. This is similar to web color pickers but formatted slightly different. Most HEX color pickers use a hashtag in the front of the value, like, **#000000**. In CircuitPython, instead of a hashtag, **0x** is used. Here's a few examples.

- Black = 0x000000
- White = 0xFFFFFF
- Purple = 0x8f42f4

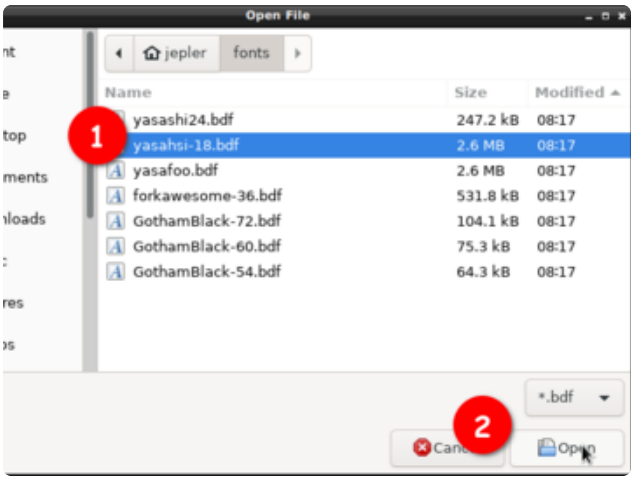
Convert to PCF

CircuitPython supports two font formats: the textual **.bdf** format and the binary **.pcf** format. By taking the extra step of converting your font to **.pcf** you make fonts load faster and also typically save some storage space on the board **CIRCUITPY** flash drive.

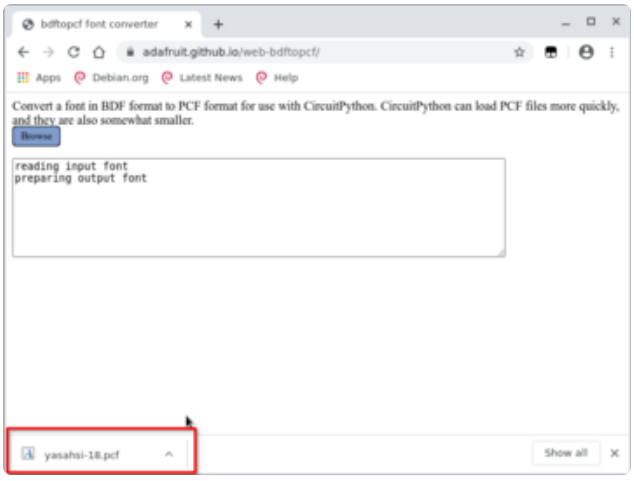
The converter software is hosted on github.io. Thanks to technology called emscripten (https://adafru.it/PEn), it runs entirely in your web browser—the font file is not uploaded to a server, which also makes it really quick. web-bdftopcf is derived from the classic font converter of the same name, a program for Unix/Linux systems. If you're interested, you can browse the C source on github (https://adafru.it/PEo).



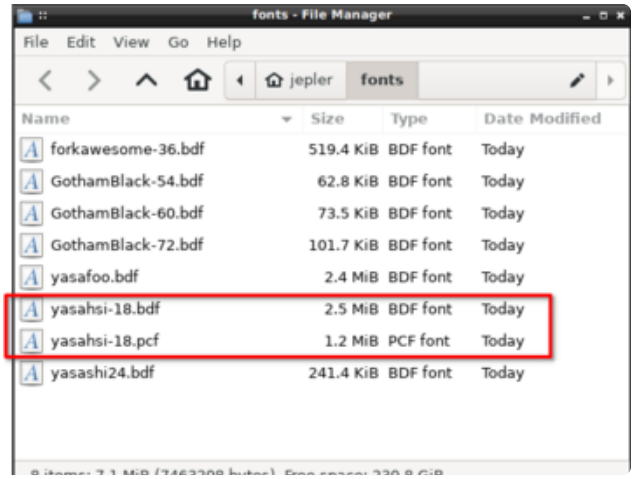
Head to https://adafruit.github.io/web-bdftopcf/ (https://adafru.it/PEp) and click the "Browse" button.



Select a **.bdf** file from your computer and click Open.



After a moment, the font will be prepared in **.pcf** format and depending on your browser settings it may be automatically downloaded or you may have to confirm that you want to download the file.



In this particular case, the **.pcf** version of the font is only half the size of the **.bdf** font, which leaves more space on the **CIRCUITPY** drive for other assets like sound files and bitmaps.

Label Updating

Sometimes when you make labels, you will want to change the text that they are displaying at some point after they've been initialized. A common example of this would be showing the value being read from a sensor, then updating it as you make new readings from the sensor.

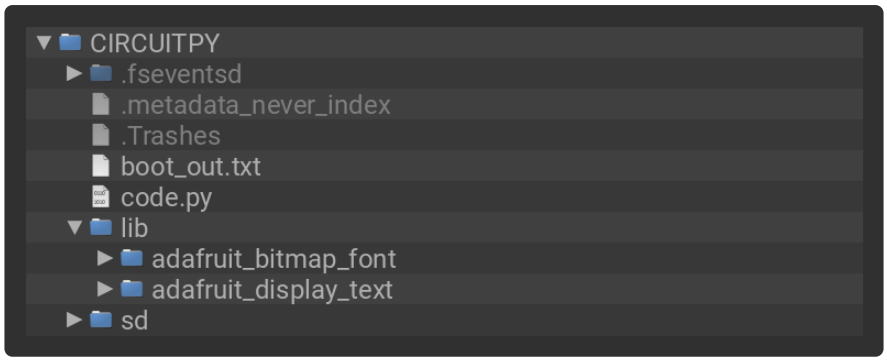
Basic Example

This example initializes a label before the main loop begins. Then inside the main loop it updates the `text` property on the label with the value it gets from `time.monotonic()`. This example can be run on any CircuitPython device with a built-in display.

Updating the text property will change what appears in the label on the display.

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **updating_text_example/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2021 Tim C, written for Adafruit Industries
#
# SPDX-License-Identifier: MIT
"""
This examples shows how to update your label with new text.
"""
import time
import board
import displayio
import terminalio
from adafruit_display_text import label

# built-in display
display = board.DISPLAY

# Make the display context
main_group = displayio.Group()
display.root_group = main_group

# create the label
updating_label = label.Label(
    font=terminalio.FONT, text="Time Is:\n{}".format(time.monotonic()), scale=2
)

# set label position on the display
updating_label.anchor_point = (0, 0)
updating_label.anchored_position = (20, 20)

# add label to group that is showing on display
main_group.append(updating_label)

# Main loop
while True:

    # update text property to change the text showing on the display
    updating_label.text = "Time Is:\n{}".format(time.monotonic())

    time.sleep(1)
```

What NOT To Do

Be careful not to create new labels infinitely in the main loop. It can lead to a Memory Error caused by making too many new labels.

Do NOT use code like below.
The code presented here is an example of the type of code you should avoid. It is not indented for you to run this code.

```
# SPDX-FileCopyrightText: 2020 Tim C, written for Adafruit Industries
#
# SPDX-License-Identifier: MIT
"""
This examples shows the WRONG way change your displayed text in the main loop.
"""
import time
import board
import displayio
import terminalio
from adafruit_display_text import label

# built-in display
display = board.DISPLAY

# Make the display context
main_group = displayio.Group()
display.root_group = main_group


# Main loop
while True:

    # You should not create labels in an unrestriced manner like this.
    #
    # DONT DO THIS!!!
    #
    new_label = label.Label(
        font=terminalio.FONT,
        text="Time Is:\n{}".format(time.monotonic()),
        scale=2,
        x=20, y=50
    )
    main_group.append(new_label)
    main_group.pop(0)

    time.sleep(1)
```