CircuitPython-GFX Reference (rev: 1.2b)

Introduction

This document provides users of the CircuitPython-GFX module: CPtGFX.py a functional reference for the user interface. The CPtGFX.py (hereafter called GFX) module is a Python language port of the Adafruit_GFX.cpp module which supports various TFT color displays from within the Arduino environment – that is, programs developed in C/C++. However, with the introduction of Circuitpython, from Adafruit, a Python version is sorely required because the user community expects as much. However, my experience after having purchased the Adafruit ItsyBitsy M4 board and the ILI9341 TFT Color Display (2.4" 240 X 320 resolution), was that there really wasn't support for anything but basic text display using a woefully small 5 X 8 default font that couldn't be changed. Also, the graphic functions we came to expect, like drawing circle, triangles and rectangles was, likewise, missing. So, I reached out to Adafruit on their forum and was told that support was "in the works" but no timeline was provided, nor was the ILI9341 or ST7735 displays high on the priority list.

Feeling like I was hyped into getting excited about the prospect of programming my M4 in Python only to then find out that I couldn't do much more than print to the serial port led me to take the bull by the horns and write this module. Admittedly, having more integrated support for graphics in Circuitpython and having the ability to utilize the 8-bit data facility is better because it's much faster; however, having "something" is better than nothing even if it's a little slower. GFX (CPtGFX.py) is my contribution to the community that allows the users of the ILI9341 and ST7735 displays to output graphics and text (larger than 5 X 8) via converted TTF fonts. This module should support any display that is a part of the Adafruit.rgb.display module. It uses the "write pixel" low level function for output. I also found that in order to output larger text, I had to either create a new, larger version of the font5x8.bin module or take a different approach and support converted TTF fonts via a "fontconvert" application that would output a font file compatible with Python (the original fontconvert program was intended to output a c/c++ .h file which was included. I modified the fontconvert.cpp program to output a Python compatible file, that is, a .py module which would be imported into the users program. So, CPtGFX.py and PYFontconvert (this is a directory name only…I kept the same program name: fontconvert) were born. In the process, I learned a bit about Freetype, refreshed my c++ knowledge, expanded my Python knowledge and learned a little more about Microsoft's Visual Studio. I built my fontconvert program using VS2010, because I was more familiar with it than with developing programs in the **ix environment – makefiles, cmakefiles, etc.

REV: 1.2b:

With this release, which is in the testing branch on github, the fontconvert program and the API CPtGFX.py module have changed in the following way. The fontconvert program will output two files: a .bin file and a .py file. The .bin file contains the font bitmap data; the .py file contains the glyphs and font related constants. The glyphs, as before, contain a bitmap offset

point to the start of the bitmap for the character, but instead of the bitmap being a part of the .py file and making it much larger, they are in a separate, more compact file. The CPtGFX.py module now reads the bitmap file as each character is rendered. Since the bitmaps are no longer a part of the .py file, it has shrunk by orders of magnitude (2-3times). Since the .bin file contains the bitmap data only, without the other characters (0x, and "," and "\n" and spaces) that were previously written out. The net result of these changes is a reduction in memory requirements, especially for large font files (e.g. 24PT). The nature of these changes dictates that the new font file pair be coupled with the new API module CPtGFX.py. **THESE PIECES MUST BE USED TOGETHER.**

END REV: 1.2b

So, let's get started with the original purpose of this document and describe the user interface by listing each available function provided. Functions are listed in alphabetical order making it easier to find what's required. However, I first describe how to perform basic setup in the Python program.

Include your converted font file and CPtGFX modules

In order to make your converted font file available to display text, you must have a statement like the following near the top of your Python program:

```
import CPtGFX as CPtGFX
import ariali12pt7b as ariali12pt7b
from Adafruit_rgb_display import color565
import ili9341 # removed from Adafruit_rgb_display library
import rgb     # removed from Adafruit_rgb_display library
import busio
import board
import time
import digitalio
cs = digitalio.DigitalInOut(board.D2)
dc = digitalio.DigitalInOut(board.D7)
# Setup hardware SPI at 32mhz on ESP8266 MicroPython.
spi = busio.SPI(clock=board.SCK, MOSI=board.MOSI,
MISO=board.MISO)
# Setup ILI9341 display using TFT CS & DC pins.
display = ili9341.ILI9341(spi, cs=cs, dc=dc)
gfx = CPtGFX.GFX(240, 320, pixel=display.pixel,
font_name=ariali12pt7b, dispobj=display,
newWidth=display.newWidth,
```

```
newHeight=display.newHeight)  # rev. 10/8 1.1b
```

NOTE: with rev: 1.2b the .bin file must be on the CircuitPython drive and its name is the same as the .py file with the exception of the file extension being .bin.

As you can see from the above, we're including several modules. The "ariali12pt7b" is a converted italic 12pt arial TTF font file, the Adafruit_rgb_display is the base module for the various TFT displays in Circuitpython. We're using SPI, because that's all that's supported so far and we're creating two objects: "display" to talk directly to the ILI9341 (this is where the "pixel()" function is defined, and the gfx object which is where the graphics functions are defined. One point: you could fill the screen either with display.fill(color) or gfx.setFill(color) – either will work equally well.

A note about fonts:

The default 5 x 8 font considers the character "baseline" position to be the upper-left corner of the character's boundary box (an imaginary box which surrounds each character); whereas TTF (or Freetype) fonts consider the "baseline" position to be at the **lower-left** corner of the character's boundary box. This is important to remember when you decide the cursor position at which you want to begin your text. One thing that might throw you is that if you issue a setCurcor(0, 0) and then do a getCursor(), you will find that the y value is different that what you specified. The reason for that is because with custom fonts, the y value must be offset based on the minimum y advance value + 1 in order for data to appear at the top of the screen if you specify (0, 0) as your starting point. Previously, I put the burden on the programmer to twiddle with the y value until it came out right. In the latest version of the program, I modified the fontconvert.exe and CPtGFX programs so that you no longer have to do any twiddling (unless twiddling is your thing ☺). You can take a look at the glyphs in the converted font file and see that the last value in each glyph is a negative number which is the "y – offset" value. If you look at the new converted ariali12pt7b.py font file, you'll see a new value down near the bottom: GFXMinYadvance. This is the minimum of all the yadvance values in the glyphs to which I add 1 to get the correct y value after a setCursor(x, y) is issued. The GFXyadvance value that's found in the font file is, essentially, the line-to-line spacing if you were to write continuously to the screen, as in the "graphicstest." example program.

Function Reference

gfx = GFX(swidth, sheight, pixel_name, font_name, dispobj=display, newWidthFunction, newHeightFunction)

Creates a graphics object (class name is GFX) which contains all the functions and variables associated with the graphics object. Importantly, it contains a pointer (dispobj) to the ili9341 instance object. This is used to improve speed by taking advantage of driver low level code.

draw_char(char, x, y)

Draws a character on the screen at the (x, y) coordinates provided. The foreground color, and size are the values previously set with the appropriate set… function(setTextColor(), setTextSize()). Note: that when writing text, the background color will correspond to the current background at that (x, y) position.

drawCircle(x, y, r, color)

Draws a circle of radius r pixels with its center being at (x, y) with the color specified.

drawFastHLine(x, y, w, color)

Draws a horizontal line, starting at (x, y) for a width of w pixels in the specified color.

drawFastVLine(x, y, h, color)

Draws a vertical line, starting at (x, y) for a height of h pixels in the specified color.

drawLine(x0, y0, x1, y1, color)

Draws a line from position (x0, y0) to position (x1, y1) in the specified color.

drawRect(x, y, w, h, color)

Draws a rectangle whose upper left corner is at (x, y) and whose width is w and height is h in the specified color.

drawRoundRect(x, y, w, y, r, color)

Draws a rectangle with rounded corners whose upper left corner is at (x, y) with a width of w, a height of h and of the specified color. The radius value is specified in pixels and represents the portion of a circle of radius r at each corner.

drawTriangle(x0, y0, x1, y1, x2, y2, color)

Draws a triangle with sides (x0, y0), (x1, y1); (x1, y1), (x2, y2); (x2, y2), (x0, y0). So, three lines are drawn between each successive set of coordinates.

fillCircle(x, y, r, color)

Draws a circle whose center is at coordinates (x, y) with a radius of r and fill it with the specified color.

fillRect(x0, y0, w, h, color)

Draws a rectangle at position (x0, y0) and filled with the specified color

fillWRect(x0, y0, w, h, color)

Draws a rectangle at position (x0, y0) as above, but uses a different algorithm which is faster when the width is significantly greater than the height.

fillTriangle(x0, y0, x1, y1, x2, y2, color)

Draws a triangle whose inside will be the specified color. See drawTriangle() for more info.

fillRoundRect(x, y, w, y, r, color)

Draws a round rectangle at the (x, y) coordinates provided with a width of w and a height of h. The rectangle will be filled with the color value. The corners will be rounded in accordance with the r value. See drawRoundRect() for more info.

fcvar = getTextColor()

Puts the value of the current text foreground color value into the variable fcvar. The default text foreground color is WHITE.

bcvar = getBgColor()

Puts the value of the current background color value into the variable bcvar. The default text background color is BLACK.

var = getTextSize()

Puts the current value of the text size variable into var. The default text size is 1.

vx, vy = getCursor()

Puts the current value of the x and y cursor values into the vx and vy variables.

cx = getCursorX()

Puts the current value of the x cursor value into the cx variable.

cy = getCursorY()

Puts the current value of the y cursor value into the cy variable.

bmap = getBitmap(char)

Gets the bitmap values for a custom font for the specified char and places the values into the tuple bmap. There are a variable number of items retrieved depending on the character.

wp = getTextWrap()

Puts the current value of the text wrap value (a Boolean) in the variable wp. The default value is True.

setWrapErase(Boolean value)

This function is used to specify what happens when text overflows in the y direction. If this value is set to True, then the screen will be erased when the overflow occurs, and then writing of text continues. If this property is not set (i.e. = False) then when an overflow occurs in the y direction, a filled rectangle will be written at the top of the screen, overwriting whatever text was there first, and then the new text will be written.

setTextSize(newsize)

Sets newsize as the current text size. Note: the newsize value can be 1, 2 or 3. However, I strongly suggest that instead of setting text size to a number larger than 1, create a converted font file with a larger size, issues setFont(newfontfile), write your text, then setFont(prevfont) back your original value. The text from a converted font file is much more appealing to look at than the text created by setting to a larger text size value, plus it renders quicker. If you preview the font file you're converting you can see the available sizes. So, if you've got a scriptbl 12 pt, do another convert but specify 24pt to get a font file with larger text. Then you can switch back and forth at will.

we = getWrapErase()

Retrieves the current setting of the WrapErase property and places it in the we variable specified.

x, y, w, h = getTextBounds('some text', x, y)

Obtains the coordinates of the rectangle that will enclose the text if written at the provided (x, y) point. The width of the rectangle is returned as the w variable and the height is returned as the h variable. So, if you want to write some highlighted text (text with a different colored background) use getTextBounds('some text', x, y) to get the upper-left corner coordinates and the width and height of the rectangle; then, issue a fillRectangle(x, y, w, h, newbgcolor) and then issue a text('some text', x, y). The end result of these 3 function calls will be highlighted text with a different colored background.

setCursor(x, y)

Sets the current CursorX and CursorY values to the supplied x and y values. Remember, if you do a setCursor(x, y) and then immediately issue a getCursor(), you'll find that your original y value has been changed in accordance with the font file GFXMinYadvance value contained therein. Continue using the adjusted value. Whenever you issue the setCursor(x, y) this adjustment will be made. Only the y values for text are changed, the y values used in the graphics figure functions are not changed. Also, if you're using the default font5x8.bin font file, no adjustment is necessary.

setBgColor(bgcolor)

Sets the current background color to the value of bgcolor. The color value should be obtained by using the color565(r, g, b) function provided in the Adafruit_rgb_display module. Default is:

BLACK.

setTextColor(fgcolor)

Sets the current foreground color to the value of fgcolor. The color value should be obtained by using the color565(r, g, b) function of Adafruit_rgb_display. Default is WHITE.

setTextWrap(wv)

Sets the current text wrap Boolean value to the supplied wv value, either True of False to indicate what happens when text being written will exceed the boundaries of the display. If text wrap is off and text exceeds the boundary, it will disappear; if text wrap is on and text being written will exceed the screen boundaries, it will wrap to the next line. If at the bottom of the screen, it will wrap back to the top.

text(str, x, y)

Display the text represented by the string str on the display at the given (x, y) coordinates. The text will be written in the current foreground color as specified in the most recent setTextColor() or the default (WHITE). After the text has been written, issue:

"x, y = getCursor()" to get the updated cursor position

txtw = width('some text')

Returns the text width of the string specified as 'some text.' Note: this function is valid ONLY when using the default font5x8.bin. If using a custom font use the getTextBounds(str, x, y) function to obtain the text size values. The default font is 5 pixels high.

writeLine(x0, y0, x1, y1, color)

Draw a line from point (x0, y0) to (x1, y1) in the specified color.

writeFillRect(x, y, w, h, color)

Draw a filled rectangle whose upper-left corner is at point (x, y) and whose width is w, height is h and fill it with the specified color.

Users will note some functions that provide the same result. The only reason I can give for this is that I tried to duplicate the relevant functions from the Adafruit_GFX.cpp program. I'm suspecting that some of these apparent duplicate functions were put there to avoid having to re-write some other code. For example: writeRectangle() and fillRect() do exactly the same thing in exactly the same way. My suggestion is for users to use the fillRect() function rather than the writeRectangle as it's name reflects the result. Future releases will probably eliminate these duplicate functions as they take up valuable program space.

setFill(color)

This is essentially a duplicate (in fact it calls) of the fill(color) function of the ILI9341 display object. I felt it's cleaner if all calls were made to the GFX object, instead of the ILI9341 object.

setRotation(val)

This function handles screen rotation. Valid values of val are: 0, 1, 2 or 3. A value of 0 places the screen in "normal" mode with the SD card at the top-right. Rotation mode 1 places the screen such that the SD card slot is on the left side, towards the top. Rotation mode 2 places the screen such that the SD card slot is at the bottom towards the left. Rotation mode 3 places the screen such that the SD card slot is on the right side, towards the bottom. Note that in each screen orientation, the (0, 0) point is at the top-left. Text is written left to right, top to bottom. Modes 0 and 2 are "portrait" mode, modes 1 and 3 are "landscape" mode.

setScroll(dy)

Scroll the display upwards, dy number of pixels. If you want to continuously scroll the screen, place this function in a loop…see the "graphicstext.py" in the examples directory of the repository.