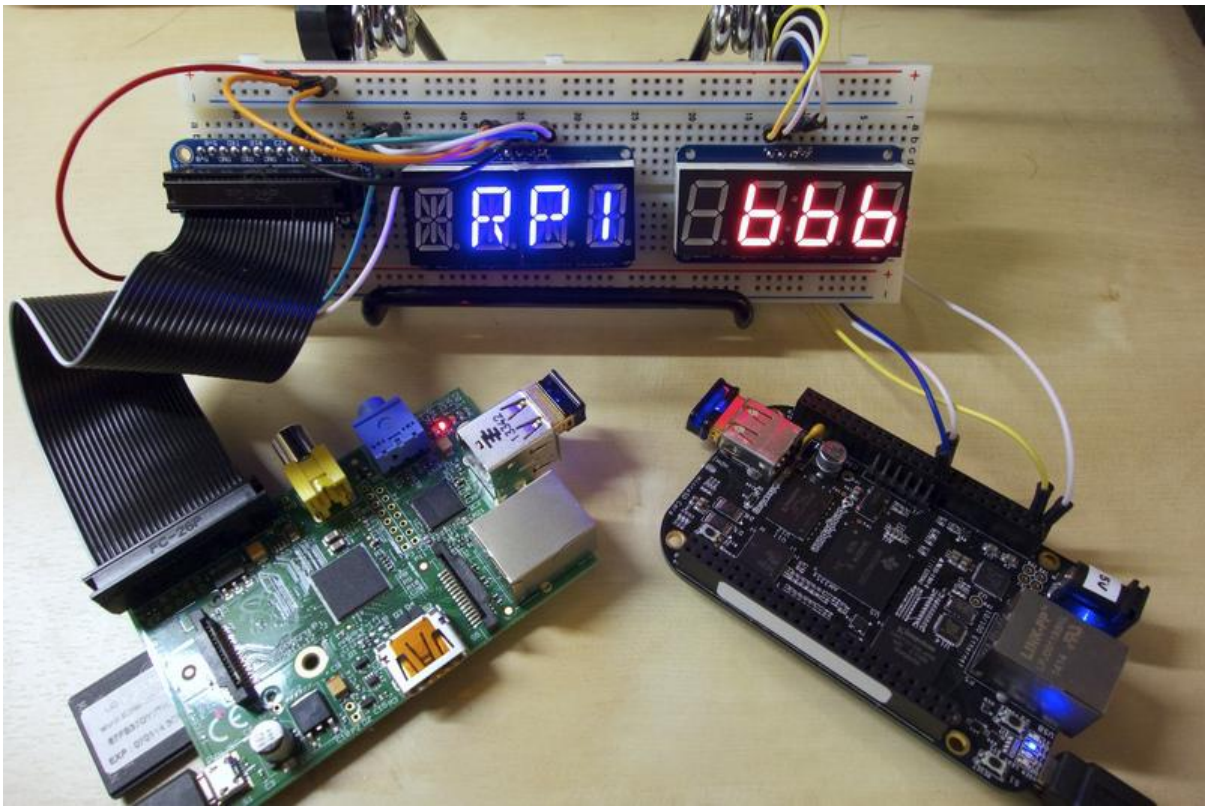
LED Backpack Displays on Raspberry Pi and BeagleBone Black

Created by Tony DiCola



https://learn.adafruit.com/led-backpack-displays-on-raspberry-pi-and-beaglebone-black

Last updated on 2023-08-29 02:34:51 PM EDT

Table of Contents

Overview

The code examples in this guide are no longer supported. Please see the guide "Matrix and 7-Segment LED Backpack with the Raspberry Pi" for the latest examples. We recommend using the CircuitPython HT16K33 library to interface with the LED Backpack Displays.

Recommended Guide: Matrix and 7-Segment LED Backpack with the Raspberry Pi ()

LED backpack displays are a great way to add a simple, bright LED display to your project. These displays get their name because of the controller chip attached to the back of them like a 'backpack'. This HT16K33 controller can drive up to 128 multiplexed LEDs in matrix, bar graph, 7-segment numeric, and even 14-segment alpha-numeric configurations. It handles the LEDs with a constant-current driver so the light is bright and consistent even if the power supply varies.

Best of all, the controller exposes a very simple I2C-based interface which is easy for a small-board computer like the Raspberry Pi or BeagleBone Black to access. This guide will show you how to install and use the Adafruit Python LED backpack library () for driving LED backpack displays from a Raspberry Pi or BeagleBone Black!

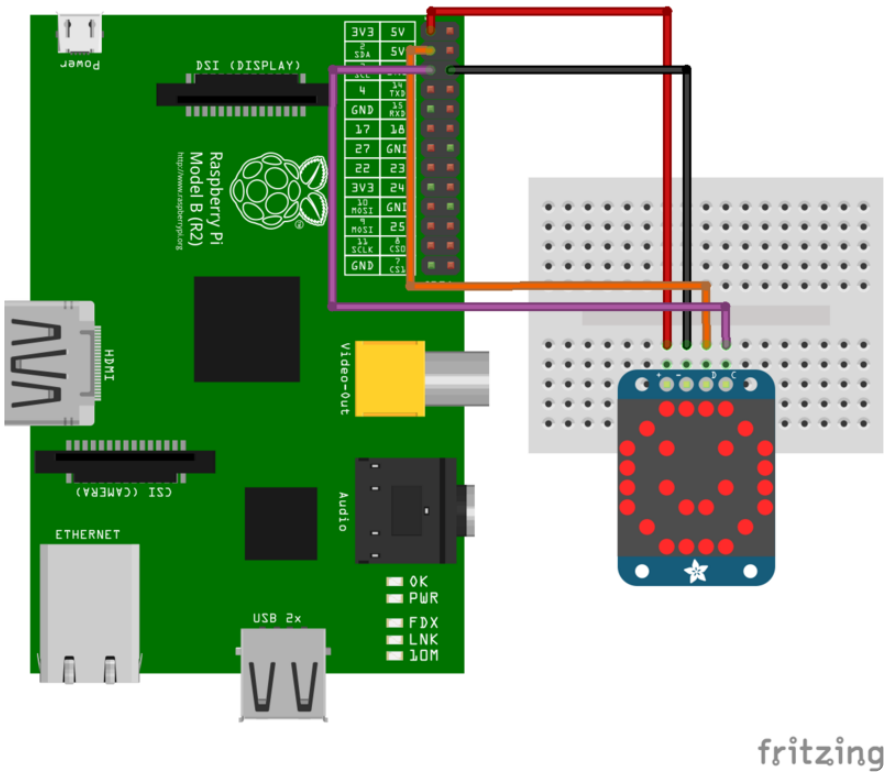
Before you get started be sure to read the LED backpack guide () to familiarize yourself with the assembly and basic usage of the backpack displays. You will also want to be familiar with connecting to and using a Raspberry Pi () or BeagleBone Black's Linux command line (). Finally, make sure your Raspberry Pi is running the latest Raspbian () or Occidentalis () operating system, or your BeagleBone Black is running the official Debian () or a Debian-based operating system like Ubuntu.

Wiring

Raspberry Pi

Note: Before wiring up your display, make sure you've enabled I2C access on the Raspberry Pi ().

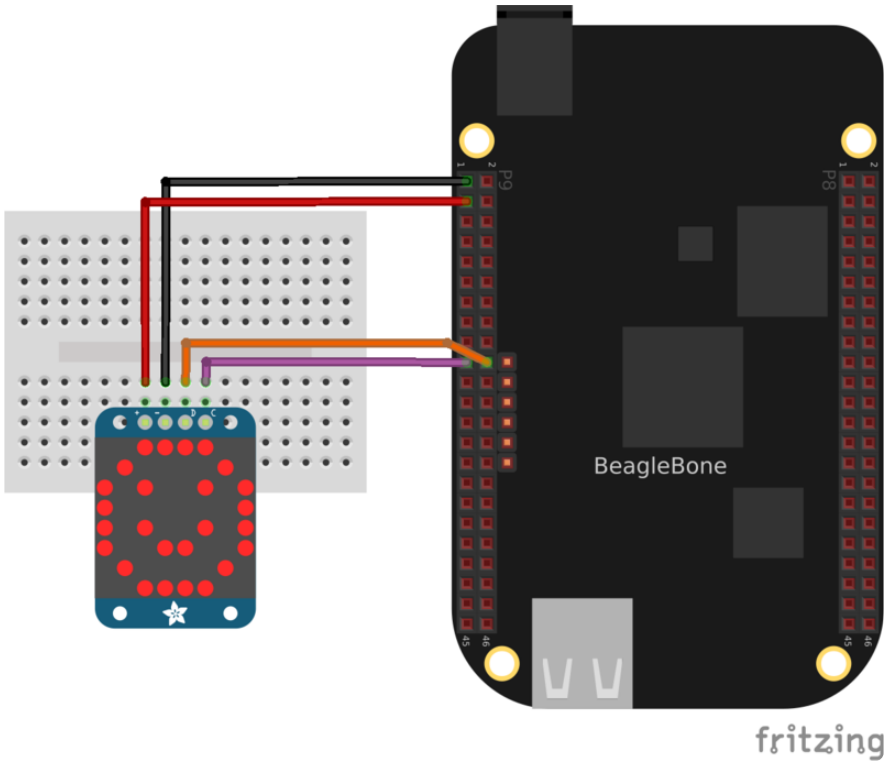
Wire up the LED backpack display to the Pi as follows (note the image below shows a matrix display, however the wiring is the same for any backpack):



- Connect display + (power) to Raspberry Pi 3.3V or 5V power (red wire). 5V is brighter but if you have other devices on the I2C bus its better to go with 3.3V
- Connect display - (ground) to Raspberry Pi ground (black wire).
- Connect display D (data/SDA) to Raspberry Pi SDA (orange wire).
- Connect display C (clock/SCL) to Raspberry Pi SCL (purple wire).
- If there's a Vi2c or IO pin, connect that to 3.3V as well

BeagleBone Black

Wire up the LED backpack display to the BeagleBone Black as follows (note the image below shows a matrix display, however the wiring is the same for any backpack):



- Connect display + (power) to BeagleBone Black 3.3V or 5V power (red wire). 5V is brighter but if you have other devices on the I2C bus its better to go with 3.3V
- Connect display - (ground) to BeagleBone Black ground (black wire).
- Connect display D (data/SDA) to BeagleBone Black I2C2_SDA pin P9_20 (orange wire).
- Connect display C (clock/SCL) to BeagleBone Black I2C2_SCL pin P9_19 (purple wire).
- If there's a Vi2c or IO pin, connect that to 3.3V as well

Note that the BeagleBone Black has two I2C interfaces and this wiring will use the/dev/i2c-1 interface. Make sure there aren't any device tree overlays loaded () which use these I2C pins for other purposes. The default BeagleBone Black device tree configuration with no overlays loaded will expose the necessary I2C interface for the wiring above.

Usage

Installation

Before installing the LED backpack Python library you must first install a few dependencies. Make sure your Raspberry Pi or BeagleBone Black has internet access, then connect to its command prompt and execute:

```
sudo apt-get update
sudo apt-get install build-essential python-dev python-smbus python-imaging git
```

Next download the LED backpack Python library code and examples () by executing:

```
git clone https://github.com/adafruit/Adafruit_Python_LED_Backpack.git
cd Adafruit_Python_LED_Backpack
sudo python setup.py install
```

After executing the commands above the LED backpack Python library will be installed and available for any Python script to use.

Usage

You can find a few examples of the library's usage inside the examples subdirectory. Navigate into this directory and run the appropriate script for your display. For example if you have your device wired to an 8x8 matrix, you would execute:

```
cd examples
sudo python matrix8x8_test.py
```

Once running, you should see different pixels of the matrix light up as the display cycles through lighting all the pixels individually. Then the example should end by displaying a graphic of a square with an X in the middle on the display.

I'll walk through the matrix8x8_test.py code below to describe the important usage details of the LED backpack library:

```
import time

import Image
import ImageDraw

from Adafruit_LED_Backpack import Matrix8x8
```

The first line imports the standard Python time library which is needed for the sleep function used later in the script. The next lines import the Python Imaging Library () which is used for drawing shapes on the display.

The final line imports the Matrix8x8 module from the Adafruit_LED_Backpack package. This line is important because it tells Python that the script will need to use the LED backpack library.

```
# Create display instance on default I2C address (0x70) and bus number.
display = Matrix8x8.Matrix8x8()

# Alternatively, create a display with a specific I2C address and/or bus.
```

```
# display = Matrix8x8.Matrix8x8(address=0x74, busnum=1)

# Initialize the display. Must be called once before using the display.
display.begin()
```

The next lines create an instance of the Matrix8x8 class and initialize it by calling the begin() method. You must call begin() once before calling other functions to update the display!

Below the first line is a commented line which shows how to create a Matrix8x8 class that uses a different I2C address or bus number. If you're using multiple displays on the same I2C bus they all must have a unique I2C address. By soldering jumpers on the back of a backpack display closed () you can control the address associated with a display. This commented line shows how to pass an address parameter which overrides the default 0x70 address with a new value of 0x74.

If you need to use a different I2C bus, for example on a BeagleBone Black which has multiple I2C buses, you can pass the desired bus number in the busnum parameter. If you don't provide this busnum parameter the library will try to determine the default I2C bus for your device--if you followed the wiring in this guide you should be all set and not need to provide the I2C bus number.

```
# Run through each pixel individually and turn it on.
for x in range(8):
    for y in range(8):
        # Clear the display buffer.
        display.clear()
        # Set pixel at position i, j to on.  To turn off a pixel set
        # the last parameter to 0.
        display.set_pixel(x, y, 1)
        # Write the display buffer to the hardware.  This must be called to
        # update the actual display LEDs.
        display.write_display()
        # Delay for half a second.
        time.sleep(0.5)
```

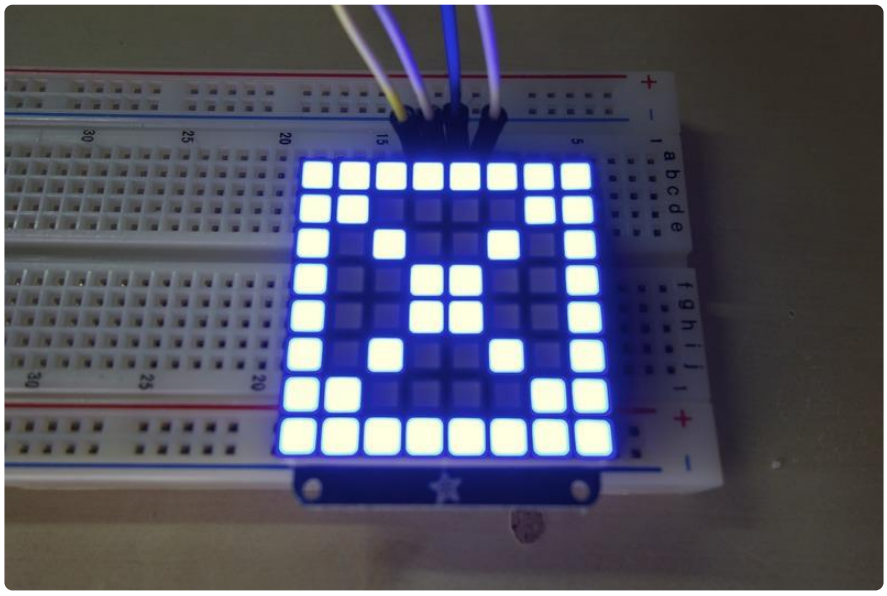
The next part of the program will iterate through all 8 rows and columns of the display and light up each pixel individually. You can see the clear() function is first used to turn off all the LEDs in the display buffer. Then the set_pixel() function is called and given the x, y coordinates of the pixel and a value of 1 to enable the pixel. Finally the write_display() function is called to write the display buffer to the hardware. It's very important to remember to call write_display() or else you won't see the display LEDs change!

```
# Draw some shapes using the Python Imaging Library.

# Clear the display buffer.
display.clear()

# First create an 8x8 1 bit color image.
image = Image.new('1', (8, 8))
```

```
# Then create a draw instance.
draw = ImageDraw.Draw(image)

# Draw a rectangle with colored outline
draw.rectangle((0,0,7,7), outline=255, fill=0)

# Draw an X with two lines.
draw.line((1,1,6,6), fill=255)
draw.line((1,6,6,1), fill=255)

# Draw the image on the display buffer.
display.set_image(image)

# Draw the buffer to the display hardware.
display.write_display()
```

The final part of the program uses the Python Imaging Library () to draw shapes on the display. First the display buffer is cleared, and then an 8x8 pixel 1-bit color image is created. Once the image is created a drawing class () is created which allows drawing on the image buffer. You can see how the rectangle and line functions draw basic shapes on the image. Finally the set_image() function is called to write the image to the display buffer, and the display buffer is written to the hardware with write_display( ).

If you'd like to draw text on the display, check out the SSD1306 library examples () for a sample of drawing text with the PIL library. However the 8x8 display is small and can't display more than a character or two!



Above you can see an example of graphics drawn on the 8x8 matrix display.

To use other displays such as the bicolor matrix, bicolor bar graph, 7 segment, or 14 segment display the basic usage is very similar to the Matrix8x8 class above. You must first import the associated module from the Adafruit_LED_Backpack package, create an instance of the class (optionally passing a different I2C address or bus

number), and call the begin() function to initialize the display.

Beyond the setup, each display class has slightly different functionality which I'll cover below.

Bicolor Matrix 8x8

The bicolor matrix display is demonstrated in the bicolor_matrix8x8_test.py script in the examples folder. This code demonstrates using the BicolorMatrix8x8 class in a very similar way to the Matrix8x8 class above. The key difference is that the set_pixel( ) function takes a color value instead of a 1 or 0 value like with the Matrix8x8 class.

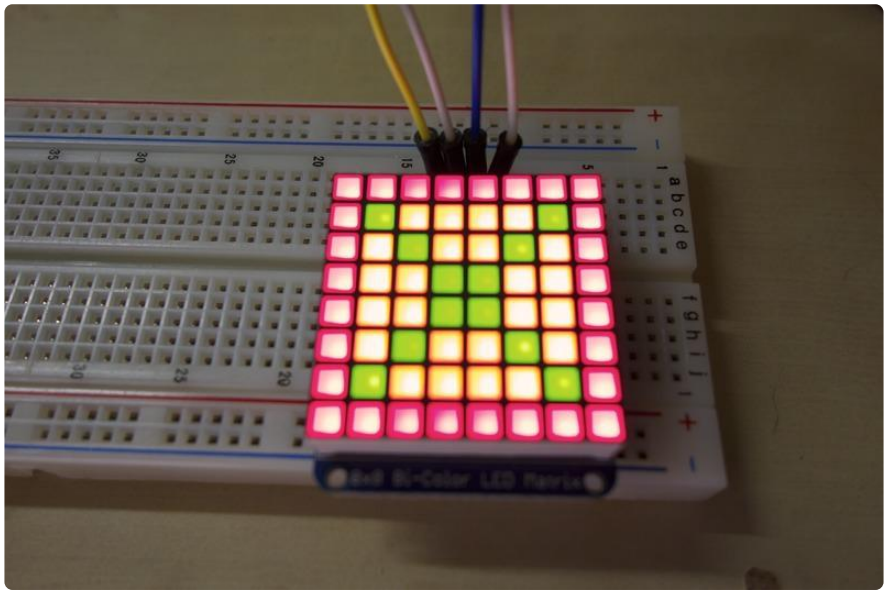
For example to set the pixel at position 1,1 red, the pixel at position 1,2 green, and the pixel at position 1,3 yellow you could write:

```
display.set_pixel(1, 1, BicolorMatrix8x8.RED)
display.set_pixel(1, 2, BicolorMatrix8x8.GREEN)
display.set_pixel(1, 3, BicolorMatrix8x8.YELLOW)
```

Notice that each color is represented by a global value in the BicolorMatrix8x8 module. You can also pass a value of 0 to turn off a specific LED.

Also the Python Imaging Library is slightly different than with the single color display. Instead of creating a 1-bit color image a full RGB color image is created. Any pixel in the image that is red (i.e. red component is 255 and all other components are 0) will have a red LED, and likewise for green (green component is 255) or yellow (red and green components are 255). Any other color pixel will be treated as no color/LED lit.

Below you can see an example of graphics on the bicolor 8x8 matrix:



Bicolor Bar Graph 24

The bicolor bar graph display is demonstrated in the bicolor_bargraph24_test.py scri pt in the examples folder. This code will light up 3 bars at a time with different colors and move them up the display. In addition to animating the bars, the example will also cycle through the 16 different brightness levels that can be set for any of the LED backpack displays.

The main loop of the example looks like:

```
# Run through all bars and colors at different brightness levels.
print 'Press Ctrl-C to quit.'
brightness = 15
while True:
    # Set display brightness (15 is max, 0 is min).
    display.set_brightness(brightness)
    for i in range(24):
        # Clear the display buffer.
        display.clear()
        # Light up 3 bars, each a different color and position.
        display.set_bar(i,   BicolorBargraph24.RED)
        display.set_bar(i+1, BicolorBargraph24.GREEN)
        display.set_bar(i+2, BicolorBargraph24.YELLOW)
        # Write the display buffer to the hardware.  This must be called to
        # update the actual display LEDs.
        display.write_display()
        # Delay for half a second.
        time.sleep(0.5)
    # Decrease brightness, wrapping back to 15 if necessary.
    brightness -= 1
    if brightness == 0:
        brightness = 15
```

You can see the code starts at brightness level 15, the maximum brightness level, by calling the set_brightness() function. With each iteration of the animation the code will decrease the brightness by one level, until it reaches 0 (minimum brightness) and wraps back up to max brightness.

It's important to note all of the LED backpack classes support the set_brightness() function! You can dim all the pixels in a matrix, bargraph, or segment display.

You can also see the BicolorBargraph24 class has a set_bar() function which turns on a bar at the specified position (0 to 23) with the given color. Just like with the Bicolor Matrix8x8 class you specify the color from a global value in the module.

7 Segment

The 7 segment numeric display is demonstrated in the sevensegment_test.py script in the examples folder. If you run this example it will loop through displaying different numeric values with different precision and justification (left or right side of the

display). The example will also show printing hexadecimal values in a simple counter.

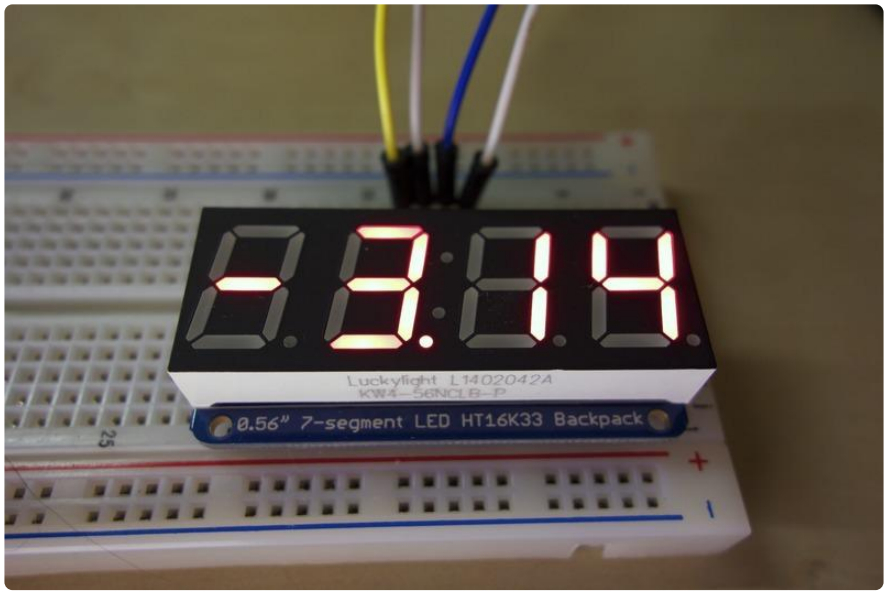
The important functions with the 7 segment display are the print_float() and print_hex( ) functions. The print_float() function will attempt to print a numeric value (either integer or floating point/fractional) to the 4 character 7 segment display. Your value must be small enough to fit on the display, or else you'll see an error display of 4 dashes, ----. For negative values a negative sign will be printed, but remember it takes up an entire digit to display!

With the print_float() function you can control how many decimal digits are printed for a floating point/fractional value. By default 2 decimal digits will be displayed, however you can specify the decimal_digits parameter with a different number of digits (including 0 for no decimal digits!). The printed value will be rounded to the desired precision and printed on the display.

In addition to controlling the decimal digit precision, you can also change the justification of the number on the display. By default numbers are printed with right justification, which means they are closest to the right side of the display. However if you pass the justify_right=False parameter to the function then the number will be displayed closest to the left side of the display.

The print_hex() function will take an integer value between 0 and FFFF and display it in hexadecimal. This is useful for displaying a large range of values like a counter, however it's less intuitive to read (unless you're a machine that understands hexadecimal!). The print_hex() function also has an optional justify_right boolean parameter which controls which side of the display to print the value towards.

Below you can see an example of the 7 segment display showing the number -3.14:



14 Segment Alphanumeric Display

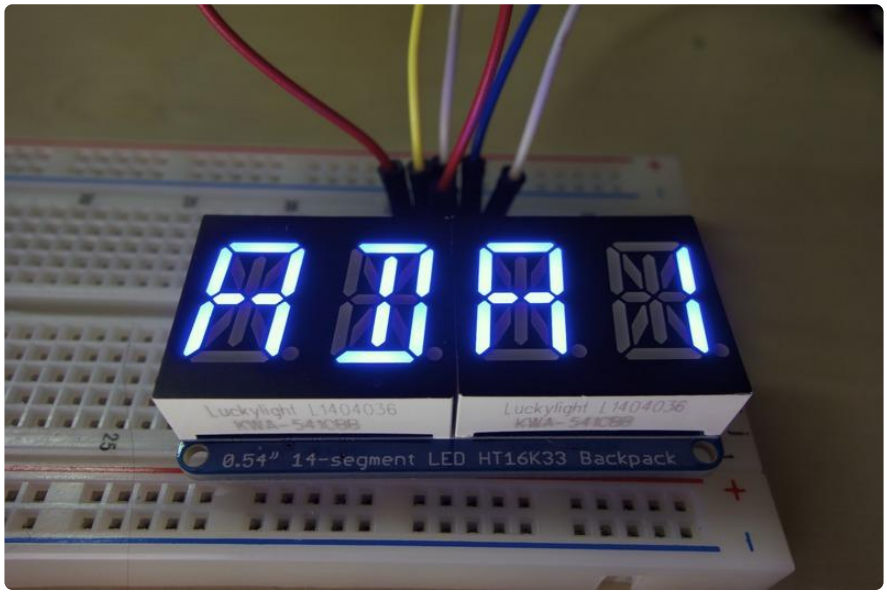
The 14 segment alphanumeric display is demonstrated in the alphanum4_test.py script in the examples folder. If you run this example it will scroll a text message across the display. This display is great for showing complete text messages because the 14 segment displays are very flexible.

With the 14 segment display the usage is very similar to the 7 segment display above. Just like the 7 segment display you can use the print_float() and print_hex() functions to print numeric and hexadecimal values. However in addition to those functions you can also use the print_str() function to print any 4 character printable ASCII string!

For example to print 'ADA!' (without quotes) to the display you could call:

```
display.print_str('ADA!')
```

Using the 14-segment display is as easy as that, just call print_str() with 4 characters of text! Note that only the ASCII values 32-127 (the printable ASCII character range) are supported by the library right now.



Above you can see the 14-segment display with the text 'ADA!' written to it.

Enjoy using your LED backpack displays with the Python library! If you find issues or would like to contribute to the code, feel free to do so on the library's GitHub home ().