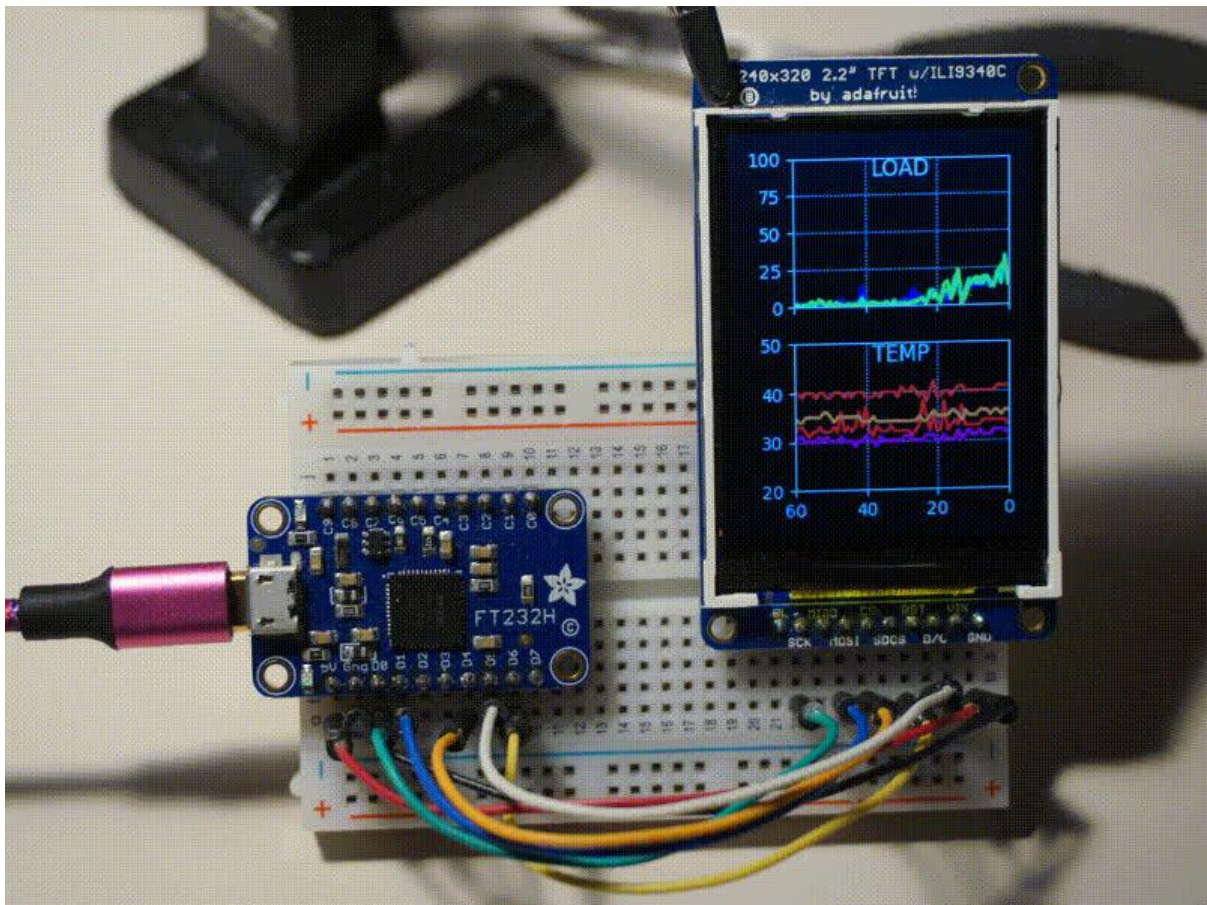




Desktop or Laptop TFT Sidekick With FT232H

Created by Carter Nelson



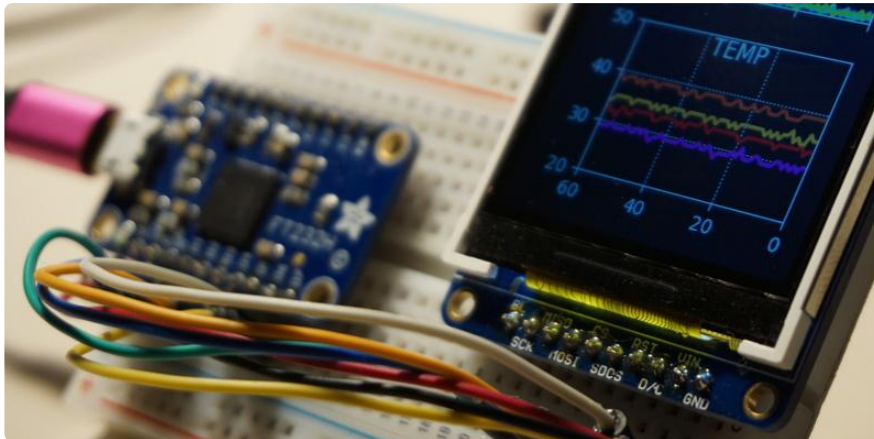
<https://learn.adafruit.com/tft-sidekick-with-ft232h>

Last updated on 2021-11-15 07:52:48 PM EST

Table of Contents

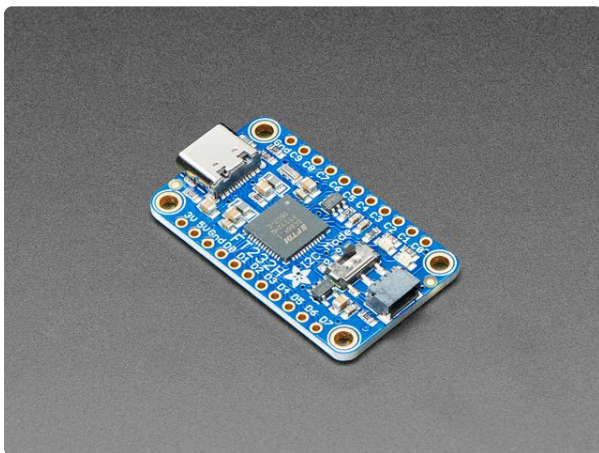
Overview	3
Hardware Setup	4
Software Setup	5
• Python	5
• FT232H Setup	5
• Matplotlib	6
• Pillow/PIL	6
• psutil	6
Basic Test	6
How It Works	9
• Initial Setup	10
• Update Data	11
• Update Plots	11
• Goto 2	12
• Customization	12
CPU Load	15
Memory Usage	18
Network Traffic	21

Overview



The [FT232H breakout \(https://adafru.it/xhf\)](https://adafru.it/xhf) board allows you to add I2C, SPI, and GPIO ports to [almost any PC with a USB port \(https://adafru.it/FWD\)](https://adafru.it/FWD). The SPI port capability allows for driving SPI based graphical TFT displays. And by running on a PC, you have plenty of processing power and access to the entire Python ecosystem. This combo allows for some fun use cases.

In this guide we will show you how to use this approach to display real time system status using an FT232H breakout and a small TFT display. It will be your little system status Sidekick.



[Adafruit FT232H Breakout - General Purpose USB to GPIO, SPI, I2C](https://adafruit.com/product/2264)

Wouldn't it be cool to drive a tiny OLED display, read a <https://www.adafruit.com/product/2264>

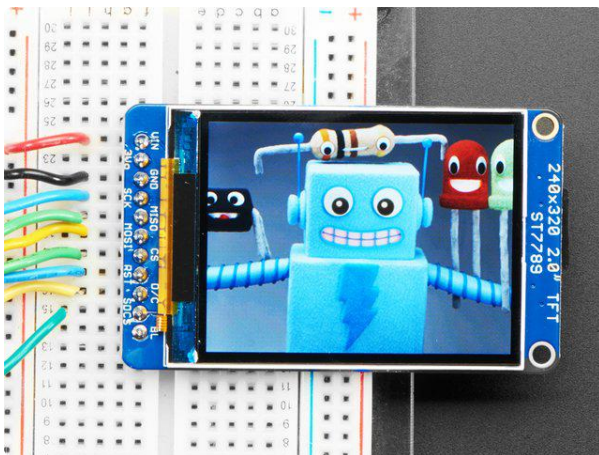
You can use just about any of our TFT, OLED or E-Ink displays as we've got Python drivers for all of them. However, we recommend one of our color TFT's cause they update fast and look great with nice bright colors



2.2" 18-bit color TFT LCD display with microSD card breakout

This lovely little display breakout is the best way to add a small, colorful and bright display to any project. Since the display uses 4-wire SPI to communicate and has its own...

<https://www.adafruit.com/product/1480>



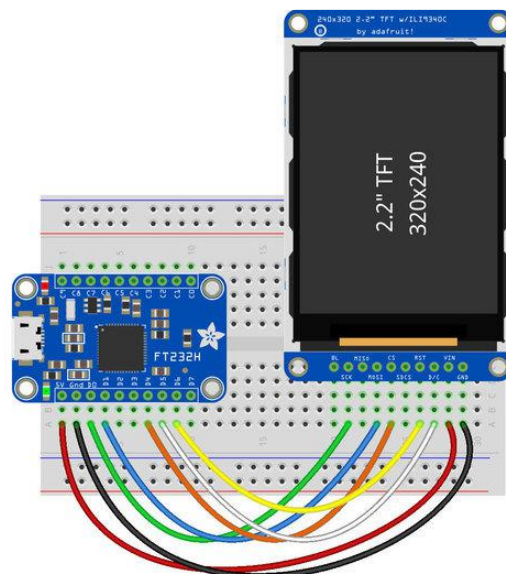
2.0" 320x240 Color IPS TFT Display with microSD Card Breakout

This gorgeous IPS display breakout is the best way to add a small, colorful and bright display to any project, with excellent visibility from any angle. Since the display uses 4-wire...

<https://www.adafruit.com/product/4311>

Hardware Setup

Here's the wiring diagram for connecting the FT232H to the TFT display. A breadboard setup is fine for getting started, but it might be nice to package this up prettier if you want to really Sidekick it up.



The FT232H to TFT wiring is as follows:

- 5V to VIN
- GND to GND
- D0 to SCK
- D1 to MOSI
- D4 to CS
- D5 to D/C
- D6 to RST

If you're using a different display than the 2.2" TFT make sure its in SPI mode and follow the descriptive wiring table above

Software Setup

We will need several software components to get this all working. Here's the run down.

Python

You may already have this installed. If not, here's the official website you can go to for install info:

Python Main Page

<https://adafru.it/deW>

To check if it's already installed, try running this at a command line:

```
python3 --version
```

We are showing running with explicit `python3` just to make sure you run with Python 3 instead of 2. Don't use Python 2.

FT232H Setup

This is covered in the guide here:

FT232H Setup

<https://adafru.it/GNe>

Don't forget to set the BLINKA_FT232H environment variable so that the Adafruit Blinka library knows it should probe the USB bus to find an FT232H chip. This is OS specific, so see details in the guide.

Matplotlib

This is used to generate plots. Use the official installation instructions here to install for your platform:

Matplotlib Install

<https://adafru.it/GNf>

Pillow/PIL

This is the Python Imaging Library which is used to render the plots onto bitmaps that can be sent to the TFT display. Install instructions are here:

Pillow Install

<https://adafru.it/FUT>

psutil

This is a Python library used to gather system data. The PyPi page is [here \(https://adafru.it/GNA\)](https://adafru.it/GNA) and the install instructions are here:

psutil Install

<https://adafru.it/GNB>

Basic Test

Before we get to plotting various system status parameters, let's use a stand alone test to make sure all the essential software pieces are working. We'll also use this example as a way to explain how the code works. If you wanted to adapt this to plot something else, this would be a good starting point.

Here's the code:

```

import time
from collections import deque
import random
import math
# Blinka CircuitPython
import board
import digitalio
import adafruit_rgb_display.ili9341 as ili9341
# Matplotlib
import matplotlib.pyplot as plt
# Python Imaging Library
from PIL import Image

#pylint: disable=bad-continuation
#==| User Config |=====
REFRESH_RATE = 0.5
HIST_SIZE = 61
PLOT_CONFIG = (
    #-----
    # PLOT 1 (upper plot)
    #-----
    {
        'line_config' : (
            {'color' : '#FF0000', 'width' : 2, 'style' : '--'},
            {}),
        },
    #-----
    # PLOT 2 (lower plot)
    #-----
    {
        'title' : 'sin()',
        'ylim' : (-1.5, 1.5),
        'line_config' : (
            {'color' : '#00FF00', 'width' : 4},
            )
    }
)

def update_data():
    ''' Do whatever to update your data here. General form is:
        ... y_data[plot][line].append(new_data_point)
        ...
    # upper plot data
    for data in y_data[0]:
        data.append(random.random())

    # lower plot data
    y_data[1][0].append(math.sin(0.5 * time.monotonic()))

#==| User Config |=====
#pylint: enable=bad-continuation

# Setup X data storage
x_time = [x * REFRESH_RATE for x in range(HIST_SIZE)]
x_time.reverse()

# Setup Y data storage
y_data = [ [deque([None] * HIST_SIZE, maxlen=HIST_SIZE) for _ in
            plot['line_config']]
            for plot in PLOT_CONFIG
            ]

# Setup display
disp = ili9341.ILI9341(board.SPI(), baudrate = 24000000,
                       cs = digitalio.DigitalInOut(board.D4),
                       dc = digitalio.DigitalInOut(board.D5),
                       rst = digitalio.DigitalInOut(board.D6))

```

```

# Setup plot figure
plt.style.use('dark_background')
fig, ax = plt.subplots(2, 1, figsize=(disp.width / 100, disp.height / 100))

# Setup plot axis
ax[0].xaxis.set_ticklabels([])
for plot, a in enumerate(ax):
    # add grid to all plots
    a.grid(True, linestyle=':')
    # limit and invert x time axis
    a.set_xlim(min(x_time), max(x_time))
    a.invert_xaxis()
    # custom settings
    if 'title' in PLOT_CONFIG[plot]:
        a.set_title(PLOT_CONFIG[plot]['title'], position=(0.5, 0.8))
    if 'ylim' in PLOT_CONFIG[plot]:
        a.set_ylim(PLOT_CONFIG[plot]['ylim'])

# Setup plot lines
#pylint: disable=redefined-outer-name
plot_lines = []
for plot, config in enumerate(PLOT_CONFIG):
    lines = []
    for index, line_config in enumerate(config['line_config']):
        # create line
        line, = ax[plot].plot(x_time, y_data[plot][index])
        # custom settings
        if 'color' in line_config:
            line.set_color(line_config['color'])
        if 'width' in line_config:
            line.set_linewidth(line_config['width'])
        if 'style' in line_config:
            line.set_linestyle(line_config['style'])
        # add line to list
        lines.append(line)
    plot_lines.append(lines)

def update_plot():
    # update lines with latest data
    for plot, lines in enumerate(plot_lines):
        for index, line in enumerate(lines):
            line.set_ydata(y_data[plot][index])
        # autoscale if not specified
        if 'ylim' not in PLOT_CONFIG[plot].keys():
            ax[plot].relim()
            ax[plot].autoscale_view()
    # draw the plots
    canvas = plt.get_current_fig_manager().canvas
    plt.tight_layout()
    canvas.draw()
    # transfer into PIL image and display
    image = Image.frombytes('RGB', canvas.get_width_height(),
                            canvas.tostring_rgb())
    disp.image(image)

print("looping")
while True:
    update_data()
    update_plot()
    time.sleep(REFRESH_RATE)

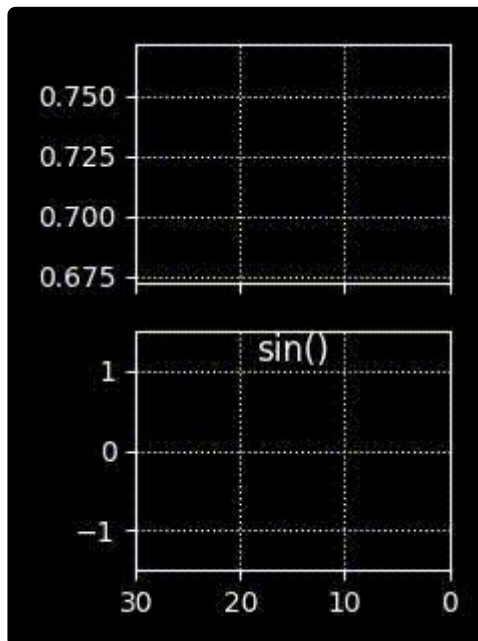
```

And then to run it do the following. If you've already set the BLINKA_FT232H environment variable, you don't need to do that again.

Make sure you've set the BLINKA_FT232H environment variable so Blinka knows to look for the FT232H on the USB bus!

```
$ export BLINKA_FT232H=1
$ python3 tft_sidekick_basic.py
```

You should get something like this on the display.



The top plot is two lines of random numbers. The bottom line is a sine curve. In the next section we'll go through this code and talk about how it works.

Also, check the [FT232H SPI page](https://adafru.it/GNF) for a simplified test for SPI TFT displays (<https://adafru.it/GNF>)

How It Works

The code basically works like this:

1. Initial setup
2. Update data
3. Update plots
4. Goto 2

Let's go through each of these.

Initial Setup

The initial setup creates the lists (actually [dequeues](https://adafru.it/GNC) (<https://adafru.it/GNC>)) the data will be stored in, the TFT setup, and the plot setup including the axis and the plot lines. This ends up being a fair amount of the code. It's all this:

```
# Setup X data storage
x_time = [x * REFRESH_RATE for x in range(HIST_SIZE)]
x_time.reverse()

# Setup Y data storage
y_data = [ [deque([None] * HIST_SIZE, maxlen=HIST_SIZE) for _ in
plot['line_config']]
            for plot in PLOT_CONFIG
            ]

# Setup display
disp = ili9341.ILI9341(board.SPI(), baudrate = 24000000,
                      cs = digitalio.DigitalInOut(board.D4),
                      dc = digitalio.DigitalInOut(board.D5),
                      rst = digitalio.DigitalInOut(board.D6))

# Setup plot figure
plt.style.use('dark_background')
fig, ax = plt.subplots(2, 1, figsize=(disp.width / 100, disp.height / 100))

# Setup plot axis
ax[0].xaxis.set_ticklabels([])
for plot, a in enumerate(ax):
    # add grid to all plots
    a.grid(True, linestyle=':')
    # limit and invert x time axis
    a.set_xlim(min(x_time), max(x_time))
    a.invert_xaxis()
    # custom settings
    if 'title' in PLOT_CONFIG[plot]:
        a.set_title(PLOT_CONFIG[plot]['title'], position=(0.5, 0.8))
    if 'ylim' in PLOT_CONFIG[plot]:
        a.set_ylim(PLOT_CONFIG[plot]['ylim'])

# Setup plot lines
#pylint: disable=redefined-outer-name
plot_lines = []
for plot, config in enumerate(PLOT_CONFIG):
    lines = []
    for index, line_config in enumerate(config['line_config']):
        # create line
        line, = ax[plot].plot(x_time, y_data[plot][index])
        # custom settings
        if 'color' in line_config:
            line.set_color(line_config['color'])
        if 'width' in line_config:
            line.set_linewidth(line_config['width'])
        if 'style' in line_config:
            line.set_linestyle(line_config['style'])
        # add line to list
        lines.append(line)
    plot_lines.append(lines)
```

Update Data

Updating the data is done in the function `update_data()`. You would change this to be whatever you want (more info below). Wherever the new data comes from, you would then just add each new data point to the data stores. For the basic example, this is just random numbers and a sine curve:

```
def update_data():
    ''' Do whatever to update your data here. General form is:
        ... y_data[plot][line].append(new_data_point)
        ...
    # upper plot data
    for data in y_data[0]:
        data.append(random.random())

    # lower plot data
    y_data[1][0].append(math.sin(0.5 * time.monotonic()))
```

Each line of the plot has a dedicated deque to store its data. Each plot stores these deques in a list. These lists are then stored in the variable `y_data`. Therefore, to access the `line` for a given `plot`, you'd use the syntax `y_data[plot][line]`. You'll generally want to use the `append()` method to add each new data point. The result is something like:

```
y_data[plot][line].append(new_data_point)
```

You can also use Python iterator syntax if that works for your scenario. That's how the random data is appended in the example above.

Also remember that indexing is 0 based. So the top plot is 0 and the bottom plot is 1.

Update Plots

Updating the plots is where Matplotlib and Pillow are used. This is all done in the function `update_plot()`. The basic idea is to update the `ydata` for each line with the current data. The plot is then re-rendered. Pillow is then used to generate an image object that can be sent to the TFT display.

You generally won't have to deal with this function.

```
def update_plot():
    # update lines with latest data
    for plot, lines in enumerate(plot_lines):
        for index, line in enumerate(lines):
            line.set_ydata(y_data[plot][index])
```

```

# autoscale if not specified
if 'ylim' not in PLOT_CONFIG[plot].keys():
    ax[plot].relim()
    ax[plot].autoscale_view()
# draw the plots
canvas = plt.get_current_fig_manager().canvas
plt.tight_layout()
canvas.draw()
# transfer into PIL image and display
image = Image.frombytes('RGB', canvas.get_width_height(),
                        canvas.tostring_rgb())
disp.image(image)

```

Goto 2

And that's it. The whole thing is driven by the simple loop at the bottom:

```

print("looping")
while True:
    update_data()
    update_plot()
    time.sleep(REFRESH_RATE)

```

Drawing to the display requires quite a bit of CPU power, so we need to add a `time.sleep()` so we don't slow down the PC. Note that if you reduce the delay, your CPU usage will increase, and there's a natural limit of perhaps 10 FPS before we can't send data fast enough to the display. Don't expect video-level playback

Customization

We skipped over a bunch of additional "initial setup" at the top of the code. This is all the code between the two comment lines:

```

#==| User Config |=====

```

This is where you'll edit the code to customize it for your use. There are two general parts:

1. Configure behavior and aesthetics via the `CONSTANTS`
2. Change the `update_data()` function for your use case

Behavior and Aesthetics

The two constants `REFRESH_RATE` and `HIST_SIZE` are pretty straight forward. They determine how often the data plot is updated and how much total data to store. The

two together define the total time window for the plot as `REFRESH_RATE * HIST_SIZE`.

The more interesting constant is the `PLOT_CONFIG` structure. As the name implies, this is how you will configure the two plots. It's a bit of a nested mess of tuples and dictionaries. But hopefully it's laid out in a way to make it easy to edit.

At a minimum, you need the `line_config` entry for each plot with one empty dictionary for each line of the plot. So the absolute minimum `PLOT_CONFIG` would look like:

```
PLOT_CONFIG = (  
#-----  
# PLOT 1 (upper plot)  
#-----  
{  
  'line_config' : (  
    { },  
  )  
},  
#-----  
# PLOT 2 (lower plot)  
#-----  
{  
  'line_config' : (  
    { },  
  )  
}  
)
```

This would create a single plot line for both the upper and lower plots with default values used. A plot line is created for each dictionary entry in `line_config`. So if you wanted to add a second plot line to the upper plot, the above would become:

```
PLOT_CONFIG = (  
#-----  
# PLOT 1 (upper plot)  
#-----  
{  
  'line_config' : (  
    { },  
    { },  
  )  
},  
#-----  
# PLOT 2 (lower plot)  
#-----  
{  
  'line_config' : (  
    { },  
  )  
}  
)
```

The only change is one additional line with an empty dictionary `{ }`.

The `PLOT_CONFIG` in the basic test example tries to demonstrate your various customization options. For any of them, if they are not specified, default values are used. So, looking at the basic test example:

```
PLOT_CONFIG = (  
    #-----  
    # PLOT 1 (upper plot)  
    #-----  
    {  
        'line_config' : (  
            {'color' : '#FF0000', 'width' : 2, 'style' : '--'},  
            {}  
        )  
    },  
    #-----  
    # PLOT 2 (lower plot)  
    #-----  
    {  
        'title' : 'sin()',  
        'ylim' : (-1.5, 1.5),  
        'line_config' : (  
            {'color' : '#00FF00', 'width' : 4},  
        )  
    }  
)
```

The upper plot has no title or y axis limits set. The y axis will autoscale. Two plot lines are setup in `line_config`. The first line has `color`, `width`, and `style` specified. The second line has nothing specified so will use default settings.

The lower plot specifies a plot `title`. This is simply some text that will be shown at the top of the plot. It also species specific y axis limits via `ylim`, therefore the y axis will not autoscale. A single plot line is setup in `line_config` with `color` and `width` specified. A default `style` will be used.

In general, you should match up the number of lines configured in `line_config` with the data that is updated in `update_data()`.

Watch out for missing {}, (), and/or ','s when editing the PLOT_CONFIG structure.

Custom Data

The `update_data()` function can be whatever you want. As mentioned above, the general syntax for a given `line` of a given `plot` is:

```
y_data[plot][line].append(new_data_point)
```

The system status examples that follow will provide more examples on how this function is used.

CPU Load

OK, now let's plot something actually fun and useful. This is where using [psutil \(https://adafru.it/GNA\)](https://adafru.it/GNA) comes in. This is an excellent cross-platform system monitoring Python library. We'll use it for a couple of examples.

First up - CPU load and temperature. Here's the updated code:

```
from collections import deque
import psutil
# Blinka CircuitPython
import board
import digitalio
import adafruit_rgb_display.ili9341 as ili9341
# Matplotlib
import matplotlib.pyplot as plt
# Python Imaging Library
from PIL import Image

#pylint: disable=bad-continuation
#==| User Config |=====
REFRESH_RATE = 1
HIST_SIZE = 61
PLOT_CONFIG = (
    #-----
    # PLOT 1 (upper plot)
    #-----
    {
        'title' : 'LOAD',
        'ylim' : (0, 100),
        'line_config' : (
            {'color' : '#0000FF', 'width' : 2},
            {'color' : '#0060FF', 'width' : 2},
            {'color' : '#00FF60', 'width' : 2},
            {'color' : '#60FF00', 'width' : 2},
        )
    },
    #-----
    # PLOT 2 (lower plot)
    #-----
    {
        'title' : 'TEMP',
        'ylim' : (20, 50),
        'line_config' : (
            {'color' : '#FF0000', 'width' : 2},
            {'color' : '#FF3000', 'width' : 2},
            {'color' : '#FF8000', 'width' : 2},
            {'color' : '#Ff0080', 'width' : 2},
        )
    }
)

CPU_COUNT = 4

def update_data():
    ''' Do whatever to update your data here. General form is:
        y_data[plot][line].append(new_data_point)
    '''
```

```

...
cpu_percs = psutil.cpu_percent(interval=REFRESH_RATE, percpu=True)
for cpu in range(CPU_COUNT):
    y_data[0][cpu].append(cpu_percs[cpu])

cpu_temps = []
for shwtemp in psutil.sensors_temperatures()['coretemp']:
    if 'Core' in shwtemp.label:
        cpu_temps.append(shwtemp.current)
for cpu in range(CPU_COUNT):
    y_data[1][cpu].append(cpu_temps[cpu])

#==| User Config |=====
#pylint: enable=bad-continuation

# Setup X data storage
x_time = [x * REFRESH_RATE for x in range(HIST_SIZE)]
x_time.reverse()

# Setup Y data storage
y_data = [ [deque([None] * HIST_SIZE, maxlen=HIST_SIZE) for _ in
plot['line_config']]
            for plot in PLOT_CONFIG
            ]

# Setup display
disp = ili9341.ILI9341(board.SPI(), baudrate = 24000000,
                        cs = digitalio.DigitalInOut(board.D4),
                        dc = digitalio.DigitalInOut(board.D5),
                        rst = digitalio.DigitalInOut(board.D6))

# Setup plot figure
plt.style.use('dark_background')
fig, ax = plt.subplots(2, 1, figsize=(disp.width / 100, disp.height / 100))

# Setup plot axis
ax[0].xaxis.set_ticklabels([])
for plot, a in enumerate(ax):
    # add grid to all plots
    a.grid(True, linestyle=':')
    # limit and invert x time axis
    a.set_xlim(min(x_time), max(x_time))
    a.invert_xaxis()
    # custom settings
    if 'title' in PLOT_CONFIG[plot]:
        a.set_title(PLOT_CONFIG[plot]['title'], position=(0.5, 0.8))
    if 'ylim' in PLOT_CONFIG[plot]:
        a.set_ylim(PLOT_CONFIG[plot]['ylim'])

# Setup plot lines
#pylint: disable=redefined-outer-name
plot_lines = []
for plot, config in enumerate(PLOT_CONFIG):
    lines = []
    for index, line_config in enumerate(config['line_config']):
        # create line
        line, = ax[plot].plot(x_time, y_data[plot][index])
        # custom settings
        if 'color' in line_config:
            line.set_color(line_config['color'])
        if 'width' in line_config:
            line.set_linewidth(line_config['width'])
        if 'style' in line_config:
            line.set_linestyle(line_config['style'])
        # add line to list
        lines.append(line)
    plot_lines.append(lines)

def update_plot():

```



```

# update lines with latest data
for plot, lines in enumerate(plot_lines):
    for index, line in enumerate(lines):
        line.set_ydata(y_data[plot][index])
    # autoscale if not specified
    if 'ylim' not in PLOT_CONFIG[plot].keys():
        ax[plot].relim()
        ax[plot].autoscale_view()
# draw the plots
canvas = plt.get_current_fig_manager().canvas
plt.tight_layout()
canvas.draw()
# transfer into PIL image and display
image = Image.frombytes('RGB', canvas.get_width_height(),
                        canvas.tostring_rgb())

disp.image(image)

print("looping")
while True:
    update_data()
    update_plot()
    # update rate controlled by psutil.cpu_percent()

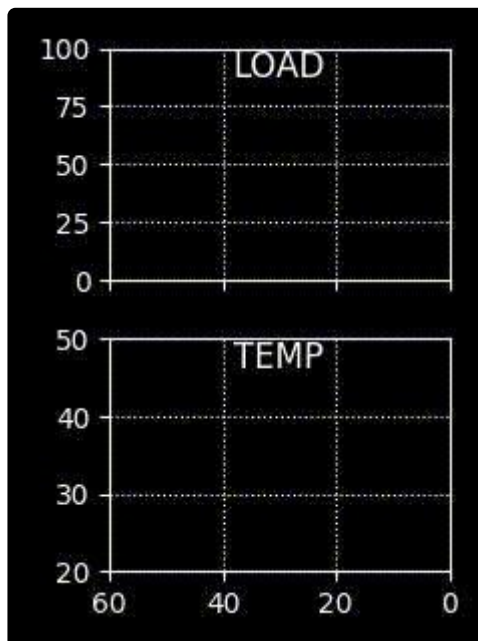
```

Save that as `tft_sidekick_cpu.py` and run it:

```
python3 tft_sidekick_cpu.py
```

Make sure you've set the `BLINKA_FT232H` environment variable.

and you should get something like this:



Note that pretty much all of the changes have been made in the user configuration section at the top. The plots are set up to track up to 4 cores. And then `update_data()` has been modified to use `psutil` to get the CPU load and temperature data and add it to the plot data.

And that's pretty much it. The one change that was done outside of this area was to remove the `time.sleep()` delay in the main loop. That's because we are using a blocking call to `psutil.cpu_percent()`, so we just let it set the pace.

Memory Usage

Next up memory usage. Precious precious memory. It comes in two precious flavors:

- virtual - the memory programs have access to (you want a lot of this)
- swap - when you run out of the above, this gets used (it's slow)

And for each of these, two things are tracked:

- free - amount of memory available
- used - amount of memory actually being used

Here's the code that tracks memory usage. It attempts to determine the total amount available on your system and use that to set the y axis limits.

```
import time
from collections import deque
import psutil
# Blinka CircuitPython
import board
import digitalio
import adafruit_rgb_display.ili9341 as ili9341
# Matplotlib
import matplotlib.pyplot as plt
# Python Imaging Library
from PIL import Image

#pylint: disable=bad-continuation
#==| User Config |=====
REFRESH_RATE = 1
HIST_SIZE = 61
PLOT_CONFIG = (
    #-----
    # PLOT 1 (upper plot)
    #-----
    {
        'title' : 'VIRTUAL',
        'ylim' : (0, psutil.virtual_memory().total / 1e9),
        'line_config' : (
            {'color' : '#00FFFF', 'width' : 2}, # free
            {'color' : '#FF00FF', 'width' : 2}, # used
        )
    }
)
```

```

    )
},
#-----
# PLOT 2 (lower plot)
#-----
{
  'title' : 'SWAP',
  'ylim' : (0, psutil.swap_memory().total / 1e9),
  'line_config' : (
    {'color' : '#00FF00', 'width' : 2}, # free
    {'color' : '#FF0000', 'width' : 2}, # used
  )
}
)
)

def update_data():
    ''' Do whatever to update your data here. General form is:
        y_data[plot][line].append(new_data_point)
    '''
    vir_mem = psutil.virtual_memory()
    y_data[0][0].append(vir_mem.free / 1e9)
    y_data[0][1].append(vir_mem.used / 1e9)

    swp_mem = psutil.swap_memory()
    y_data[1][0].append(swp_mem.free / 1e9)
    y_data[1][1].append(swp_mem.used / 1e9)

#==| User Config |=====
#pylint: enable=bad-continuation

# Setup X data storage
x_time = [x * REFRESH_RATE for x in range(HIST_SIZE)]
x_time.reverse()

# Setup Y data storage
y_data = [ [deque([None] * HIST_SIZE, maxlen=HIST_SIZE) for _ in
            plot['line_config']]
           for plot in PLOT_CONFIG
         ]

# Setup display
disp = ili9341.ILI9341(board.SPI(), baudrate = 24000000,
                      cs = digitalio.DigitalInOut(board.D4),
                      dc = digitalio.DigitalInOut(board.D5),
                      rst = digitalio.DigitalInOut(board.D6))

# Setup plot figure
plt.style.use('dark_background')
fig, ax = plt.subplots(2, 1, figsize=(disp.width / 100, disp.height / 100))

# Setup plot axis
ax[0].xaxis.set_ticklabels([])
for plot, a in enumerate(ax):
    # add grid to all plots
    a.grid(True, linestyle=':')
    # limit and invert x time axis
    a.set_xlim(min(x_time), max(x_time))
    a.invert_xaxis()
    # custom settings
    if 'title' in PLOT_CONFIG[plot]:
        a.set_title(PLOT_CONFIG[plot]['title'], position=(0.5, 0.8))
    if 'ylim' in PLOT_CONFIG[plot]:
        a.set_ylim(PLOT_CONFIG[plot]['ylim'])

# Setup plot lines
#pylint: disable=redefined-outer-name
plot_lines = []
for plot, config in enumerate(PLOT_CONFIG):
    lines = []

```

```

for index, line_config in enumerate(config['line_config']):
    # create line
    line, = ax[plot].plot(x_time, y_data[plot][index])
    # custom settings
    if 'color' in line_config:
        line.set_color(line_config['color'])
    if 'width' in line_config:
        line.set_linewidth(line_config['width'])
    if 'style' in line_config:
        line.set_linestyle(line_config['style'])
    # add line to list
    lines.append(line)
plot_lines.append(lines)

def update_plot():
    # update lines with latest data
    for plot, lines in enumerate(plot_lines):
        for index, line in enumerate(lines):
            line.set_ydata(y_data[plot][index])
        # autoscale if not specified
        if 'ylim' not in PLOT_CONFIG[plot].keys():
            ax[plot].relim()
            ax[plot].autoscale_view()
    # draw the plots
    canvas = plt.get_current_fig_manager().canvas
    plt.tight_layout()
    canvas.draw()
    # transfer into PIL image and display
    image = Image.frombytes('RGB', canvas.get_width_height(),
                           canvas.tostring_rgb())
    disp.image(image)

print("looping")
while True:
    update_data()
    update_plot()
    time.sleep(REFRESH_RATE)

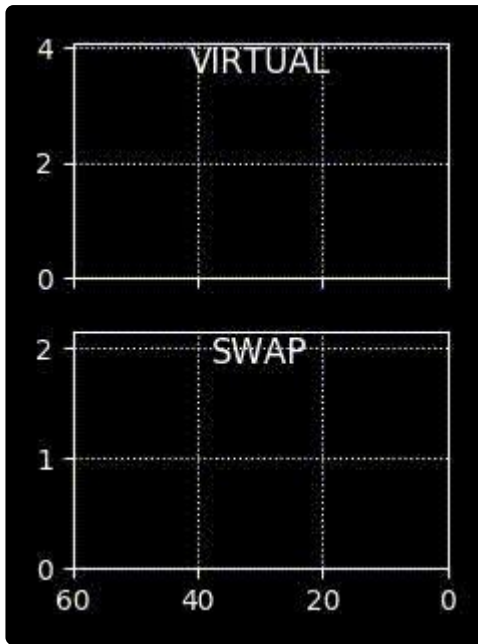
```

Save that as `tft_sidekick_mem.py` and run it with:

```
python3 tft_sidekick_mem.py
```

Make sure you've set the `BLINKA_FT232H` environment variable.

and you should get something like this:



Network Traffic

Next up, network traffic. Precious precious internet. How fast are the 1's and 0's going up and down the pipe? How much in total have gone up and down the pipe? With this example, we track both of these.

There's no easy way to pre-determine your max bandwidth. Therefore you may want to adjust the `ylim` values and the divisors in `update_data()` if you find the data too squished or going off the chart.

```
import time
from collections import deque
import psutil
# Blinka CircuitPython
import board
import digitalio
import adafruit_rgb_display.ili9341 as ili9341
# Matplotlib
import matplotlib.pyplot as plt
# Python Imaging Library
from PIL import Image

#pylint: disable=bad-continuation
#==| User Config |=====
REFRESH_RATE = 1
HIST_SIZE = 61
PLOT_CONFIG = (
    #-----
    # PLOT 1 (upper plot)
    #-----
    {
        'title' : 'RATE (MBPS)',
        'ylim' : (0, 1),
        'line_config' : (
            {'color' : '#AAFF00', 'width' : 2}, # sent
            {'color' : '#00AAFF', 'width' : 2}, # recv
        )
    }
)
```

```

    )
},
#-----
# PLOT 2 (lower plot)
#-----
{
  'title' : 'TOTAL (GB)',
  'ylim' : (0, 1),
  'line_config' : (
    {'color' : '#AAFF00', 'width' : 2}, # sent
    {'color' : '#00AAFF', 'width' : 2}, # recv
  )
}
)

def update_data():
    ''' Do whatever to update your data here. General form is:
        y_data[plot][line].append(new_data_point)
    ...
    # get two data points
    net_start = psutil.net_io_counters()
    time.sleep(REFRESH_RATE)
    net_finish = psutil.net_io_counters()

    # rate is d()/dt
    BPS_sent = (net_finish.bytes_sent - net_start.bytes_sent) / REFRESH_RATE
    BPS_recv = (net_finish.bytes_recv - net_start.bytes_recv) / REFRESH_RATE
    y_data[0][0].append(BPS_sent / 1e6)
    y_data[0][1].append(BPS_recv / 1e6)

    # total is just last
    y_data[1][0].append(net_finish.bytes_sent / 1e9)
    y_data[1][1].append(net_finish.bytes_recv / 1e9)

#==| User Config |=====
#pylint: enable=bad-continuation

# Setup X data storage
x_time = [x * REFRESH_RATE for x in range(HIST_SIZE)]
x_time.reverse()

# Setup Y data storage
y_data = [ deque([None] * HIST_SIZE, maxlen=HIST_SIZE) for _ in
           plot['line_config']]
           for plot in PLOT_CONFIG
           ]

# Setup display
disp = ili9341.ILI9341(board.SPI(), baudrate = 24000000,
                      cs = digitalio.DigitalInOut(board.D4),
                      dc = digitalio.DigitalInOut(board.D5),
                      rst = digitalio.DigitalInOut(board.D6))

# Setup plot figure
plt.style.use('dark_background')
fig, ax = plt.subplots(2, 1, figsize=(disp.width / 100, disp.height / 100))

# Setup plot axis
ax[0].xaxis.set_ticklabels([])
for plot, a in enumerate(ax):
    # add grid to all plots
    a.grid(True, linestyle=':')
    # limit and invert x time axis
    a.set_xlim(min(x_time), max(x_time))
    a.invert_xaxis()
    # custom settings
    if 'title' in PLOT_CONFIG[plot]:
        a.set_title(PLOT_CONFIG[plot]['title'], position=(0.5, 0.8))
    if 'ylim' in PLOT_CONFIG[plot]:

```

```

    a.set_ylim(PLOT_CONFIG[plot]['ylim'])

# Setup plot lines
#pylint: disable=redefined-outer-name
plot_lines = []
for plot, config in enumerate(PLOT_CONFIG):
    lines = []
    for index, line_config in enumerate(config['line_config']):
        # create line
        line, = ax[plot].plot(x_time, y_data[plot][index])
        # custom settings
        if 'color' in line_config:
            line.set_color(line_config['color'])
        if 'width' in line_config:
            line.set_linewidth(line_config['width'])
        if 'style' in line_config:
            line.set_linestyle(line_config['style'])
        # add line to list
        lines.append(line)
    plot_lines.append(lines)

def update_plot():
    # update lines with latest data
    for plot, lines in enumerate(plot_lines):
        for index, line in enumerate(lines):
            line.set_ydata(y_data[plot][index])
        # autoscale if not specified
        if 'ylim' not in PLOT_CONFIG[plot].keys():
            ax[plot].relim()
            ax[plot].autoscale_view()

    # draw the plots
    canvas = plt.get_current_fig_manager().canvas
    plt.tight_layout()
    canvas.draw()
    # transfer into PIL image and display
    image = Image.frombytes('RGB', canvas.get_width_height(),
                           canvas.tostring_rgb())

    disp.image(image)

print("looping")
while True:
    update_data()
    update_plot()
    # update rate controlled in update_data()

```

Save that as `tft_sidekick_net.py` and run it with:

```
python3 tft_sidekick_net.py
```

Make sure you've set the `BLINKA_FT232H` environment variable.

and you should get something like:

