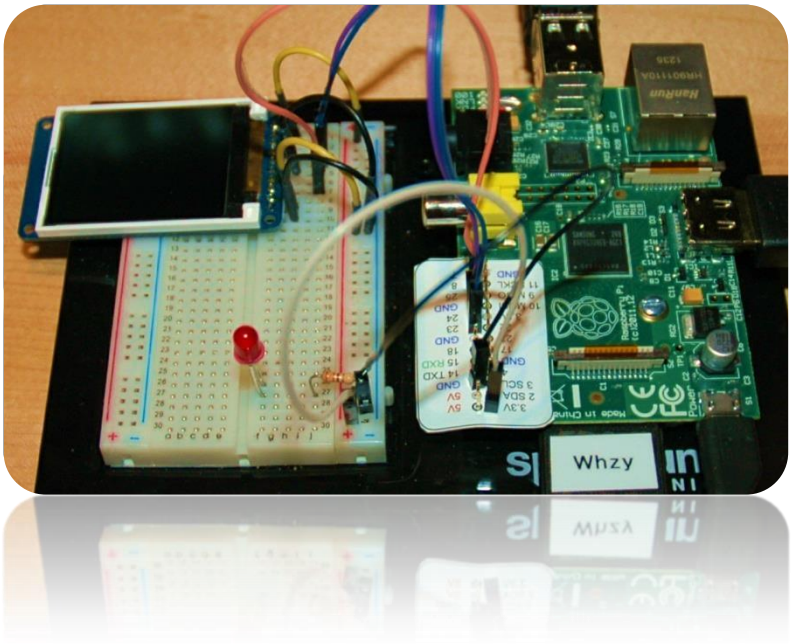


Add a TFT Display to the Raspberry Pi

Part 1: Software SPI

Bruce E. Hall, W8BH

Objective: Learn how to interface and control a 160x128 pixel TFT LCD module using Python.

1) INTRODUCTION

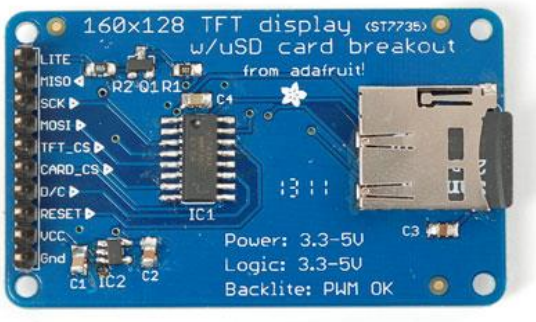
I am a big fan of 2x16 character-based LCD displays. They are simple, cheap, and readily available. Send ASCII character data to them, and they display the characters. Nice.

But sometimes characters are not enough. Colors, images, graphs, even text with fonts all require something more. The 1.8" TFT module from Adafruit (and others) gives you this option for the reasonable price of $25. Just buy one and have some fun learning what you can do with it.

This TFT module is a 128x160 LCD matrix, controlled by the onboard Sitronix ST7735 controller. You send data to it serially using the Serial-Peripheral Interface (SPI) protocol. Simple, right? Unfortunately, it's not as simple as writing to a character mode display.

I started with two documents: the Siltronix datasheet and the Adafruit library. Take a look at both of them. For me, both are bit complicated. Even the initialization code looks like a programming nightmare. I like to start simple, and build as I go. Here is my approach.

2) MAKE THE CONNECTIONS



Let's connect the hardware first. The Adafruit module has 10 pins. On the bottom of the module each pin is labeled, from pin 1 'Lite' to pin 10 'Gnd'. Mount the display module on a breadboard and connect the pins to your Raspberry Pi GPIO ports.

Male to female prototyping wires are very handy for making these point-to-point connections. Alternatively, you can bring out all of the GPIO lines to the breadboard with various third-party cables.

Here are the required connections. First, apply 3.3V power to the backlight, pin 1 and ground to pin 10. You should see the glow of the backlight when you do this. If not, check your power connections before proceeding further.

Next, connect Vcc to 3v3 and the TFT select line to Gnd.

Finally, hook up the three data lines: SCLK, MOSI, and DC. That's it!

TFT pin	Function	RPi GPIO
1	Backlight	3v3
2	MISO	
3	SCK	GPIO 24
4	MOSI	GPIO 23
5	TFT select	Gnd
6	SD select	
7	D/C	GPIO 25
8	Reset	
9	Vcc	3v3
10	Gnd	Gnd

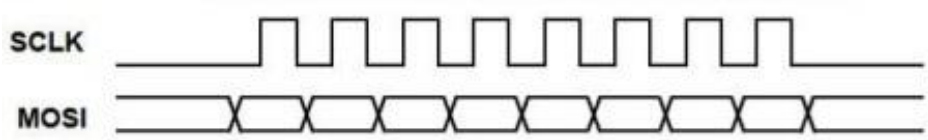
3) SERIAL PERIPHERAL INTERFACE (SPI)

Finding a good starting point is sometimes the hardest part! I chose the SPI protocol, since any data transfer to the TFT module would require this. The Pi has a built-in hardware SPI interface, which is disabled by default. I did not have any SPI peripherals to test this hardware interface, so I decided to start with a software interface instead.

At its core, the SPI algorithm is very straightforward:

- Put a data bit on the serial data line.
- Pulse the clock line.
- Repeat for all the bits you want to send, usually 8 bits at a time.

SPI is a bidirectional protocol, with two separate data lines. We need only one line, MOSI (Master-out, Slave-in) to send data from the Pi to the display. The SPI protocol also specifies different modes, depending on the active logic state of the clock line and whether the data is transferred on low-hi or hi-lo transitions. In our module the clock input SCLK is active high, and data is transferred on the low-to-high transition. This is called 'Mode 0'.



Implementing Mode 0 SPI in software is very simple. It requires two small routines, and uses two GPIO pins: one for the data and one for the clock. Here is the first routine:

```
def WriteByte(value, data=True):
        "sends byte to display using software SPI"
        mask = 0x80                             #start with bit7 (msb)
        SetPin(DC,data)                         #low = command; high = data
```

```
    for bit in range(8):                    #loop for 8 bits, msb to lsb
        SetPin(SDAT,value & mask)           #put bit on data line
        PulseClock()                        #clock in the bit
        mask >>= 1                          #go to next bit
```

The mask starts out as 0x80, which is a logic one on bit 7. The data is compared against this mask, and the GPIO line is set or reset depending on the result. For example, consider the byte value 0x61 (0b11010001):

The '&' operator performs a bitwise Boolean AND function. The result is nonzero, so the data line is set to logic 1.

Value '0x61'	1	1	0	0	0	0	0	1
Mask '0x80'	1	0	0	0	0	0	0	0
Value & Mask	1	0	0	0	0	0	0	0

Clocking the data couldn't be easier: take the clock pin high, and then return it low.

```
def PulseClock():
    "pulses the serial clock line HIGH"
    SetPin(SCLK,1)                          #bit clocked on low-high transition
    SetPin(SCLK,0)                          #no delay since python is slow
```

For each bit, the mask is shifted to the right. At the end of the loop we will have updated the GPIO data line 8 times, according to the bit values of our input data byte.

4) SENDING COMMANDS & DATA

Serial information sent to the display module can be either commands or data. For commands, the D/C (data/command) input must be 0; for data, the input must be 1. We use a third GPIO pin to supply this information. Here is the command routine:

```
WriteCmd(value):
    "Send command byte to display"
    WriteByte(value,False)
```

Our WriteByte routine has a second optional parameter, data. When omitted, data defaults to True, and the D/C pin is set to logic 1 in the WriteByte routine. We use WriteCmd to set the data parameter to false, which then sets the D/C pin to logic 0.

5) ST7735

The Sitronix ST7735 is a single-chip driver/controller for 128x160 pixel TFT-LCD displays. It can accept both serial and 8/9/16/18 bit parallel interfaces. On my module, and many others like it, only the serial interface over SPI is supported. The Sitronix datasheet refers to this interface as the four-wire SPI protocol.

The ST7735 supports over 50 different commands. Many of these commands fine-tune the power output and color intensity settings, allowing you to correct for LCD display variations. In this tutorial we need only six of those commands.

6) INITIALIZING THE DISPLAY

You should initialize the display before sending pixel data. I found a few code samples online, but they are a bit confusing. My favorite library, the Adafruit-ST7735-Library on GitHub, calls 19 different commands with over 60 parameters! Let's find an easier solution.

The first initialization step is to reset the controller, either by hardware or software. A hardware reset requires an additional GPIO line to pulse the controller's reset pin. A software reset is a byte-sized command sent to the controller. I chose the software reset, but either method should work fine. The reset function initializes the controller registers to their default values. See the reset table in datasheet section 9.14.2 for more information.

After the reset, the controller enters a low-power sleep mode. We wake the controller and turn on its TFT driver circuits with the sleep out SLPOUT command.

Finally, after turning on the driver circuits, we need to enable display output with the DISPON (display on) command. Here is the code for our simplified, 3 command routine:

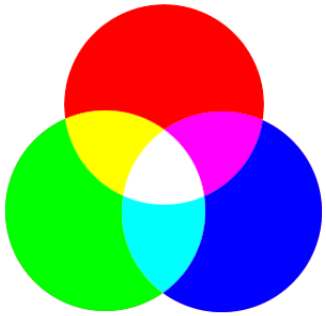
```
def InitDisplay():
      "Resets & prepares display for active use."
      WriteCmd  (SWRESET)        #software reset, puts display into sleep
      time.sleep(0.2)            #wait 200mS for controller register init
      WriteCmd  (SLPOUT)         #sleep out.
      time.sleep(0.2)            #wait 200mS for TFT driver circuits
      WriteCmd  (DISPON)         #display on!
```

7) TIME FOR COLOR

First, check your hardware connections and run a script that calls InitDisplay(). Your display should briefly blank, and then show a screen full of tiny, random color pixels. If it does, you have successfully initialized the display. If not, check your hardware connections. It's easy to get a couple GPIO pins reversed, or forget a power/ground connection. Once you get it working, it's time to send some real data.

Sometimes we need to send a single byte, sometimes we need to send a sequence of bytes. Here is a simple routine to send variable quantities of data:

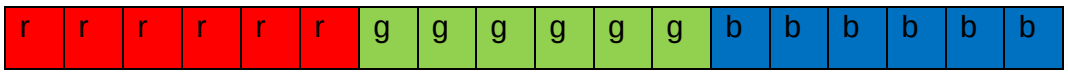
```
def WriteList (byteList):
      "Sends a list of bytes to display, as data"
      for byte in byteList:                  #grab each byte in list
            WriteByte(byte)                  #and send it
```



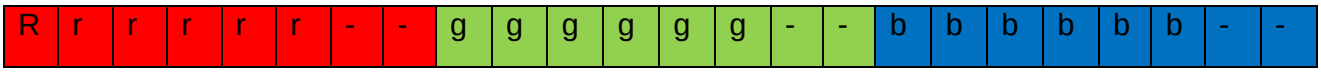
The default color mode for this controller is RGB666. Pixel colors are a combination of red, green, and blue color values. Each subcolor has 6 bits (64 different levels) of intensity. Equal amounts of red, green, and blue light produce white. Equal amounts of blue

and green produce cyan. Red and green make yellow. Since each color component is specified by 6 bits, the final color value is 18 bits in length. The number of possible color combinations in RGB666 colorspace is 2^18 = 262,144.

We represent these color combinations as an 18 bit binary number. The 6 red bits are first, followed by 6 green bits, followed by 6 blue bits:



Our controller wants to see data in byte-sized chucks, however. For every pixel we must send 24 bits (3 bytes), arranged as follows:



The lowest two bits of each red, green, and blue byte are ignored; only the 6 upper bits of each byte are used. Here is a routine to send 24 bits of pixel data:

```
def Write888(value,reps=1):
       "Sends a 24-bit RGB pixel data to display, with optional repeat"
       red = value>>16                          #red = upper 8 bits
       green = (value>>8) & 0xFF                #green = middle 8 bits
       blue = value & 0xFF                      #blue = lower 8 bits
       RGB  = [red,green,blue]                  #assemble RGB as 3 byte list
       for count in range(reps):                #send RGB value x optional repeat
              WriteList(RGB)
```

Notice how we can use right-shift and logical-and operators to select the portion of the color value corresponding to red, green, and blue. Then we build a list of the 3 bytes, and send them off. You can skip the list and call WriteByte directly, if you prefer:

```
       WriteByte(red)
       WriteByte(green)
       WriteByte(blue)
```

Some controller/display modules use BGR encoding instead of RGB. For these modules you should switch the write order of red and blue.

We must give screen coordinates before sending pixel data to the controller. The coordinates are not a single (x,y) location, but a rectangular region. To specify the region we need the controller commands CASET and RASET. The Column Address Set command sets the column boundaries, or x coordinates. The Row Address Set sets the row boundaries, or y coordinates. The two together set the display region where new data will be written.

```
def SetAddrWindow(x0,y0,x1,y1):
       "Sets a rectangular display window into which pixel data is placed"
       WriteCmd(CASET)                  #set column range (x0,x1)
       WriteWord(x0)
       WriteWord(x1)
       WriteCmd(RASET)                  #set row range (y0,y1)
       WriteWord(y0)
       WriteWord(y1)
```

We need to specify an active region, whether we're filling a large rectangle or just a single pixel. First, specify the region; next, issue a RAMWR (memory write) command; and finally, send the raw pixel data. Notice how the following routines are similar:

```
def DrawPixel(x,y,color):
        "Draws a pixel on the TFT display"
        SetAddrWindow(x,y,x,y)            #active region = 1 pixel
        WriteCmd(RAMWR)                   #memory write
        Write888(color)                   #send color for this pixel

def FillRect(x0,y0,x1,y1,color):
        "Fills a rectangle with given color"
        width = x1-x0+1                   #width of rectangle
        height = y1-y0+1                  #height of rectangle
        SetAddrWindow(x0,y0,x1,y1)        #set active region
        WriteCmd(RAMWR)                   #memory write
        Write888(color,width*height)      #send color data for all pixels
```

Both routines set the window, issue a memory write command, and then send the color data. For DrawPixel, the active window is a single pixel and only a single color value is sent. For FillRect, the active window is the entire rectangle, and the color value is sent width*height times.

In the final code below, I time how long it takes to paint the entire screen and then clear it. My Pi takes about 50 seconds to run the test. That's slow! But don't be discouraged. In Part 2 of this series we'll learn how to run the display much, much faster using hardware SPI.

8) PYTHON SCRIPT for TFT DISPLAY, PART 1:

```
#!/usr/bin/python

#############################################################################
#
#  A Python script for controlling the Adafruit 1.8" TFT LCD module
#  from a Raspbery Pi.
#
#   Author :   Bruce E. Hall, W8BH    <bhall66@gmail.com>
#   Date   :   27 Apr 2013
#
#   This module uses the ST7735 controller and SPI data interface
#
#
#   For more information, see w8bh.net
#
#############################################################################


import RPi.GPIO as GPIO
import time


#TFT to RPi connections
# PIN TFT           RPi
# 1 backlight       3v3
# 2 MISO            <none>
# 3 CLK             GPIO 24
# 4 MOSI            GPIO 23
# 5 CS-TFT          GND
# 6 CS-CARD         <none>
# 7 D/C             GPIO 25
# 8 RESET           <none>
# 9 VCC             3V3
# 10 GND            GND

SCLK    = 24
SDAT    = 23
DC      = 25
pins    = [SCLK,SDAT,DC]

#RGB888 Color constants
BLACK   = 0x000000
RED     = 0xFF0000
GREEN   = 0x00FF00
BLUE    = 0x0000FF
WHITE   = 0xFFFFFF
COLORSET = [RED,GREEN,BLUE,WHITE]


#ST7735 commands
SWRESET = 0x01                  #software reset
SLPOUT  = 0x11                  #sleep out
DISPON  = 0x29                  #display on
CASET   = 0x2A                  #column address set
RASET   = 0x2B                  #row address set
RAMWR   = 0x2C                  #RAM write
MADCTL  = 0x36                  #axis control
COLMOD  = 0x3A                  #color mode
```

```
#########################################################################
#
#   Low-level routines
#   These routines access GPIO directly
#

def SetPin(pinNumber,value):
    #sets the GPIO pin to desired value (1=on,0=off)
    GPIO.output(pinNumber,value)

def InitIO():
    GPIO.setmode(GPIO.BCM)
    GPIO.setwarnings(False)
    for pin in pins:
        GPIO.setup(pin,GPIO.OUT)
    GPIO.output(SCLK,0)                         #start with clock line low


#########################################################################
#
#   Bit-Banging (software) SPI routines:
#
#

def PulseClock():
       #pulses the serial clock line HIGH
       SetPin(SCLK,1)                           #bit clocked on low-high transition
       SetPin(SCLK,0)                           #no delay since python is slow

def WriteByte(value, data=True):
       "sends byte to display using software SPI"
       mask = 0x80                              #start with bit7 (msb)
       SetPin(DC,data)                          #low = command; high = data
       for b in range(8):                       #loop for 8 bits, msb to lsb
              SetPin(SDAT,value & mask)         #put bit on serial data line
              PulseClock()                      #clock in the bit
              mask >>= 1                        #go to next bit

def WriteCmd(value):
       "Send command byte to display"
       WriteByte(value,False)                   #set D/C line to 0 = command

def WriteWord (value):
       "sends a 16-bit word to the display as data"
       WriteByte(value >> 8)                    #write upper 8 bits
       WriteByte(value & 0xFF)                  #write lower 8 bits

def WriteList (byteList):
       "Send list of bytes to display, as data"
       for byte in byteList:                    #grab each byte in list
              WriteByte(byte)                   #and send it

def Write888(value,reps=1):
       "sends a 24-bit RGB pixel data to display, with optional repeat"
       red = value>>16                          #red = upper 8 bits
       green = (value>>8) & 0xFF                #green = middle 8 bits
       blue = value & 0xFF                      #blue = lower 8 bits
       RGB  = [red,green,blue]                  #assemble RGB as 3 byte list
       for a in range(reps):                    #send RGB x optional repeat
              WriteList(RGB)
```

```
#################################################################################
#
#   ST7735 driver routines:
#
#

def InitDisplay():
       "Resets & prepares display for active use."
       WriteCmd  (SWRESET)               #software reset, puts display into sleep
       time.sleep(0.2)                   #wait 200mS for controller register init
       WriteCmd  (SLPOUT)                #sleep out
       time.sleep(0.2)                   #wait 200mS for TFT driver circuits
       WriteCmd  (DISPON)                #display on!

def SetAddrWindow(x0,y0,x1,y1):
       "sets a rectangular display window into which pixel data is placed"
       WriteCmd(CASET)                   #set column range (x0,x1)
       WriteWord(x0)
       WriteWord(x1)
       WriteCmd(RASET)                   #set row range (y0,y1)
       WriteWord(y0)
       WriteWord(y1)

def DrawPixel(x,y,color):
       "draws a pixel on the TFT display"
       SetAddrWindow(x,y,x,y)
       WriteCmd(RAMWR)
       Write888(color)

def FillRect(x0,y0,x1,y1,color):
       "fills rectangle with given color"
       width = x1-x0+1
       height = y1-y0+1
       SetAddrWindow(x0,y0,x1,y1)
       WriteCmd(RAMWR)
       Write888(color,width*height)

def FillScreen(color):
       "Fills entire screen with given color"
       FillRect(0,0,127,159,color)

def ClearScreen():
       "Fills entire screen with black"
       FillRect(0,0,127,159,BLACK)


#################################################################################
#
#   Testing routines:
#
#

def TimeDisplay():
       "Measures time required to fill display twice"
       startTime=time.time()
       print "  Now painting screen GREEN"
       FillScreen(GREEN)
       print "  Now clearing screen"
       ClearScreen()
       elapsedTime=time.time()-startTime
       print "  Elapsed time %0.1f seconds" % (elapsedTime)
```

```
#########################################################################
#
#   Main Program
#

print "Adafruit 1.8 TFT display demo"
InitIO()
InitDisplay()
TimeDisplay()
print "Done."


#   END  ##############################################################
```