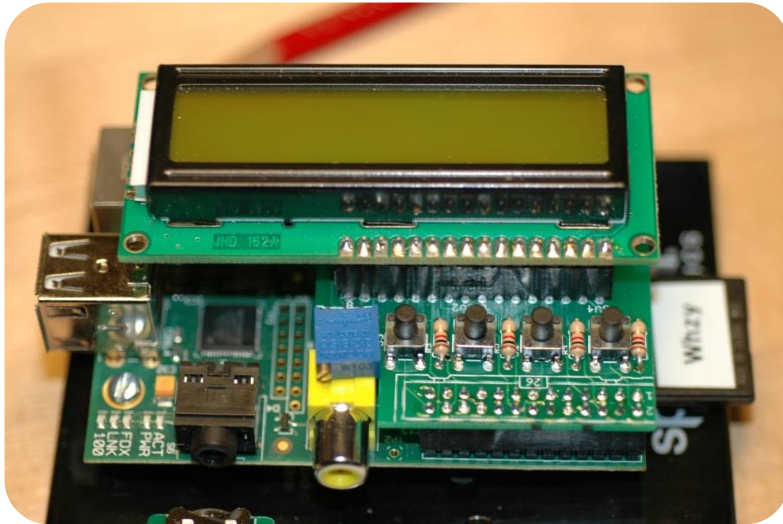


Beginner's Guide to the PI LCD

Part 4: Graphics

, W8BH

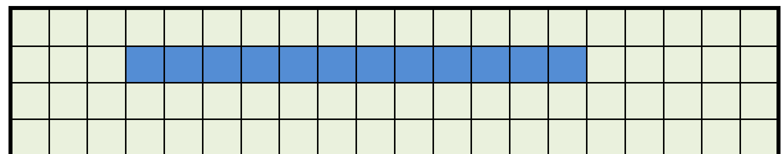
1) INTRODUCTION

In the first three parts of this series, we learned how to display text and simple graphics on the LCD board (available from mypishop.com). It's time to kick it up a notch, and create a suite of useful graphics functions. We'll even create a large-digit clock. You may have purchased either a 16x2 or a 20x4 display with your kit. In this write-up I'll be using the 20x4 display. When it comes to graphics, bigger is better!

2) SIMPLE HORIZONTAL BAR GRAPHS

An important requirement for creating detailed graphics is that the display device is dot-addressable. In other words, each display pixel can be individually addressed and programmed, separate from its neighbors. Sadly, our HD44780 controller does not give us a dot-addressable LCD display. We get only pre-determined characters, plus 8 characters of our own design. How can we possibly do artwork on that?

Well, we can't. Go ahead, prove me wrong. Detailed graphics with this LCD module are devilishly hard to do. But that doesn't mean we can't create useful, simpler graphics. Bar graphs, for example.



First, consider a single, horizontal bar graph. Here is our 20x4 display, with a horizontal bar that is 12 characters wide. If we need to display data that over a small integer range, like 0-15, we can do it by repeating the solid block (0xFF) character for the desired length. You might code it like this:

```
def HorizBar(row,startCol,length):
        GotoXY(row,startCol)                #go to starting position
        for count in range(length):
                SendByte(0xFF,True)         #display bar of desired length
```

This simple code works, and is surprisingly useful. You aren't limited to small ranges: just scale the desired range to 0-15 by the appropriate conversion factor. But your graph will always look a little coarse and chunky, since there are a limited number of possible data values/lengths.

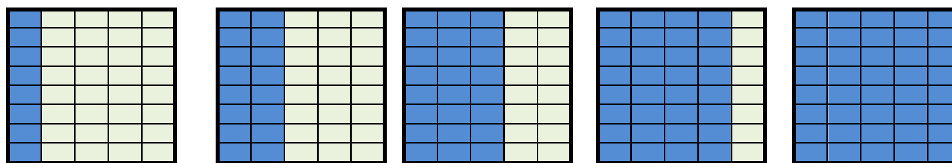
3) BETTER HORIZONTAL BAR GRAPHS

The graph will look better if we improve the horizontal resolution. But how? We can get a five-fold improvement in resolution if we use the simple graphics from Part 3.

1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0

Consider the individual character. It contains 40 individual pixel "dots", arranged in an 8 row, 5 column grid. We can't access each individual pixel, but we can create custom symbols like this vertical bar. Display this one to the right of the 12-character bar above, and you've just made a bar of length 12.2!

Let's make a set of vertical bar symbols, progressively increasing the number of columns in the symbol



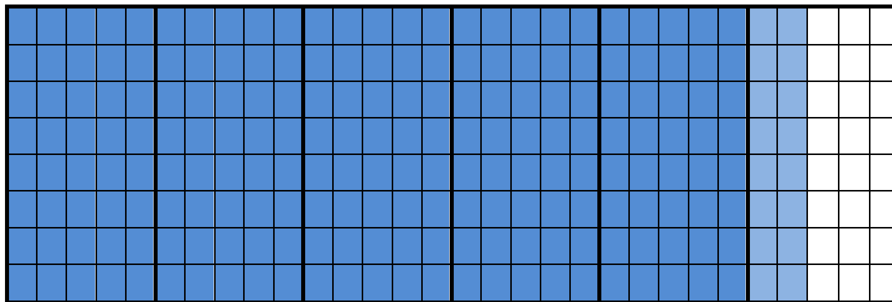
0.2 0.4 0.6 0.8 1.0

Now we can increment the length of our horizontal bar in fractions of a character, improving the horizontal resolution of our graph. It's time to code it. First, create the set of symbols, like we did in part 3:

```
horizontalBars = [
[ 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10 ],   #1 bar
[ 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18 ],   #2 bars
[ 0x1C, 0x1C, 0x1C, 0x1C, 0x1C, 0x1C, 0x1C, 0x1C ],   #3 bars
[ 0x1E, 0x1E, 0x1E, 0x1E, 0x1E, 0x1E, 0x1E, 0x1E ],   #4 bars
[ 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F ]    #5 bars
]
```

Now we need a routine to draw the horizontal bar for a given length. To simplify things, let's stay with integer lengths, and give each vertical bar a length of one (instead of 0.2). In this system our original 12 character bar is 12*5 = 60 units long. Any length can be represented

by a combination of ‘full characters’ which contain 5 bars each, followed by a terminal character containing less than 5 bars. For example, a bar graph of length 27 will be 5 full characters (5*5=25), followed by a character containing the last 2 bars:



Bar of 27 units =

5 Filled Characters + 1 Partially Filled Character

We can calculate the number of full characters by integer division: length/5. And the number of bars in the final, partially filled character is just the remainder: length % 5. A function for writing the horizontal bar sends the required number of solidly-filled characters, then a single, partially-filled character. Assume that the symbols have previously been loaded into the character-generator RAM at positions 0 through 4.

```
def DrawHBar(row,length):
       fullChars = length / 5
       bars      = length % 5
       GotoXY(row,0)                      #start at beginning of row
       for count in range(fullChars):     #full characters sent first
              SendByte(4,True)
       if bars>0:                         #final, partially filled character
              SendByte(bars-1,True)
```

The call to SendByte(4,True) sends the fourth symbol in CG-RAM, which is the 5-bar (filled) character.

4) ANIMATED HORIZONTAL BAR GRAPHS

The graphs are nice, and fun to watch a few times. But looking at fat lines gets dull after a while. Let’s spice them up a bit, and add some animation like we did in part 3 with the ‘battery charging’ symbol.

To animate the graph, we need two key functions: one to increment the graph length, and one to decrement it. Then animation becomes the simple task of keeping track of how many increments/decrements to do. Using a top-down approach, lets write this function first, and worry about the increment/decrement later.

```
def AnimatedHBar(row,startCol,newLength,oldLength=0):
diff = newLength - oldLength
for count in range(abs(diff)):
       if diff>0:
              IncrementHBar(row,startCol,oldLength)
              oldLength +=1
       else:
              DecrementHBar(row,startCol,oldLength)
              oldLength -=1
```

```
        time.sleep(ANIMATIONDELAY)
```

If the new length is greater than the old length, we increment until the new length is reached. Similarly, if the new length is less than the old length, we decrement. Each time, wait a fraction of a second for the animation effect.

Incrementing is incredibly simple: just add one bar and display it. The process of setting the cursor position and displaying a character comes up several times, so I refactored it into a little helper-function called ShowBars.

```
def ShowBars(row,col,numBars):
        GotoXY(row,col)
        if numBars==0:
                SendChar(' ')
        else:
                SendByte(numBars-1,True)

def IncrementHBar(row,length):
        #increase the number of horizontal bars by one
        fullChars = length / 5
        bars      = length % 5
        bars += 1                         #add one bar
        ShowBars(row,fullChars,bars)
```

Decrementing is a little trickier, because the number of bars in the final character cannot be allowed to go below zero . For example, a value of 25 has 5 full characters evenly with no extra bars. When one is subtracted, and the number of full characters decreases and the number of bars goes to 4:

```
def DecrementHBar(row,length):
        #reduce the number of horizontal bars by one
        fullChars = length / 5
        bars      = length % 5
        bars -= 1
        if bars<0:
                bars = 4
                fullChars -= 1
        ShowBars(row,fullChars,bars)
```

That works, but looks a bit cumbersome. Whenever things look messy, I like to play around with the code a bit. Sometimes a simpler and more intuitive solution will present itself. In this case, try doing the length decrement first, and let our integer divide and modulo functions do all the work:

```
def DecrementHBar(row,length):
        #reduce the number of horizontal bars by one
        length -= 1                      #subtract one bar
        fullChars = length / 5
        bars      = length % 5
        ShowBars(row,fullChars,bars)
```

That looks much better. Sometimes you can get ‘stuck’ because of prior assumptions and decisions. Don’t be afraid of starting over or reworking the code if it doesn’t flow the way you want.

5) VERTICAL BAR GRAPHS

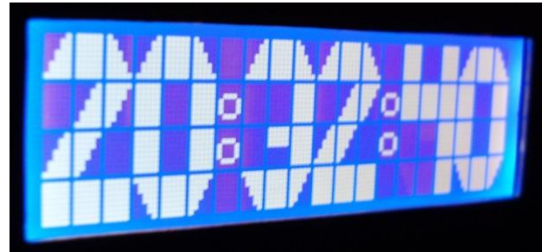
You can create vertical bar graphs in exactly the same way we did the horizontal ones:

- Create a series of vertical bar symbols
- Write routines for drawing the bar, breaking it down into full characters with a final, partially filled character
- Write an animation routine, calling on IncrementVBar and DecrementVBar

See the script listing at the end for all of the details. The code is almost exactly the same.

6) BIG CLOCK

I really enjoyed working with the Pi Matrix, because the display is fun to watch. Who doesn't like blinky LEDs? LCDs have to work harder to get the same WOW factor. I was wondering what to do for this board, when I came across some images for the "LCD smartie" clock plugin, shown here.



Now that looks like 20x4 LCD fun. But how do you program it?

Stare at the image for a while, and you'll see that each digit is rendered in a 3x4 block of characters. And each character is made up of just a few symbols: a solid block, some triangles, and some half-height blocks.

We already know how to make the custom characters. You can emulate the smartie digits with 7 different symbols: lower-right triangle, lower-left triangle, upper-right triangle, upper-left triangle, upper horizontal bar, lower horizontal bar, and solid block. Here they are:

```
digits = [
[ 0x01, 0x03, 0x03, 0x07, 0x07, 0x0F, 0x0F, 0x1F ],  #lower-rt triangle
[ 0x10, 0x18, 0x18, 0x1C, 0x1C, 0x1E, 0x1E, 0x1F ],  #lower-lf triangle
[ 0x1F, 0x0F, 0x0F, 0x07, 0x07, 0x03, 0x03, 0x01 ],  #upper-rt triangle
[ 0x1F, 0x1E, 0x1E, 0x1C, 0x1C, 0x18, 0x18, 0x10 ],  #upper-lf triangle
[ 0x00, 0x00, 0x00, 0x00, 0x1F, 0x1F, 0x1F, 0x1F ],  #lower horiz bar
[ 0x1F, 0x1F, 0x1F, 0x1F, 0x00, 0x00, 0x00, 0x00 ],  #upper horiz bar
[ 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F ]   #solid block
]
```

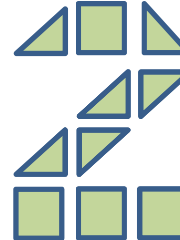
Each digit is made up of a 3x4 grid of these symbols. I created a list with 10 members for the 10 digits. Each member is a list of the 12 symbols needed to create the digit.

```
bigDigit = [
[ 0x00, 0x06, 0x01, 0x06, 0x20, 0x06, 0x06, 0x20, 0x06, 0x02, 0x06, 0x03], #0
[ 0x20, 0x06, 0x20, 0x20, 0x06, 0x20, 0x20, 0x06, 0x20, 0x20, 0x06, 0x20], #1
[ 0x00, 0x06, 0x01, 0x20, 0x00, 0x03, 0x00, 0x03, 0x20, 0x06, 0x06, 0x06], #2
[ 0x00, 0x06, 0x01, 0x20, 0x20, 0x06, 0x20, 0x05, 0x06, 0x02, 0x06, 0x03], #3
[ 0x06, 0x20, 0x06, 0x06, 0x06, 0x06, 0x20, 0x20, 0x06, 0x20, 0x20, 0x06], #4
[ 0x06, 0x06, 0x06, 0x06, 0x04, 0x04, 0x20, 0x20, 0x06, 0x06, 0x06, 0x03], #5
[ 0x00, 0x06, 0x01, 0x06, 0x20, 0x20, 0x06, 0x05, 0x01, 0x02, 0x06, 0x03], #6
[ 0x06, 0x06, 0x06, 0x20, 0x20, 0x06, 0x20, 0x20, 0x06, 0x20, 0x20, 0x06], #7
```

```
[ 0x00, 0x06, 0x01, 0x06, 0x20, 0x06, 0x06, 0x05, 0x06, 0x02, 0x06, 0x03], #8
[ 0x00, 0x06, 0x01, 0x02, 0x04, 0x06, 0x20, 0x20, 0x06, 0x20, 0x20, 0x06]  #9
]
```

0x00 indicates the first symbol (a lower-right triangle), 0x01 indicates the second (a lower-left triangle), and so on. 0x20 is a blank. The 12 symbols in each digit are ordered from top-left to lower-right.

The numeral '2' begins with a lower-right triangle (0x00) in the top left corner, followed by a solid block (0x06) and a lower-left triangle (0x01). The complete sequence is [0x00, 0x06, 0x01, 0x20, 0x00, 0x03, 0x00, 0x03, 0x20, 0x06, 0x06, 0x06]



To display a digit, take its list of 12 symbols and display them in a 3x4 matrix:

```
def ShowBigDigit(symbol,startCol):
    #displays a 4-row-high digit at specified column
    for row in range(4):
        GotoXY(row,startCol)
        for col in range(3):
            index = row*3 + col
            SendByte(symbol[index],True)
```

Time display involves getting the time digits and displaying them once a second. In the time module there is a function 'strftime' which lets us format the time. By calling with the parameter `("%I%M%S")`, a time of 12:23:56 will returned as a string '122356'.

```
def BigClock():
    #displays large-digit time in hh:mm:ss on 20x4 LCD
    LoadSymbolBlock(digits)
    posn = [0,3,7,10,14,17]  #column position for each digit
    ClearDisplay()
    ShowColon(6)
    ShowColon(13)
    while (True):                                   #CONTINOUS LOOP!
        tStr = time.strftime("%I%M%S")              #time in HHMMSS format
        for i in range(len(tStr)):                  #FOR EACH DIGIT ---
            value = int(tStr[i])                    #convert char to int
            symbols = bigDigit[value]               #get symbol list
            ShowBigDigit(symbols,posn[i])           #display digit on LCD
        time.sleep(1)                               #update clock each second
```

BigClock runs a continuous loop, so you will need to Ctrl-C from the keyboard to stop it running. Enjoy!

7) PYTHON SCRIPT for PI LCD, PART 4:

```
########################################################################
#
#    LCD4:  Learning how to control an LCD module from Pi
#
#
#
#
#    See w8bh.net for more information.
#
########################################################################

import time                     #for timing delays
import RPi.GPIO as GPIO
import random

#OUTPUTS: map GPIO to LCD lines
LCD_RS              = 7         #GPIO7  = Pi pin 26
LCD_E               = 8         #GPIO8  = Pi pin 24
LCD_D4              = 17        #GPIO17 = Pi pin 11
LCD_D5              = 18        #GPIO18 = Pi pin 12
LCD_D6              = 27        #GPIO21 = Pi pin 13
LCD_D7              = 22        #GPIO22 = Pi pin 15
OUTPUTS = [LCD_RS,LCD_E,LCD_D4,LCD_D5,LCD_D6,LCD_D7]

#HD44780 Controller Commands
CLEARDISPLAY        = 0x01
RETURNHOME          = 0x02
RIGHTTOLEFT         = 0x04
LEFTTORIGHT         = 0x06
DISPLAYOFF          = 0x08
CURSOROFF           = 0x0C
CURSORON            = 0x0E
CURSORBLINK         = 0x0F
CURSORLEFT          = 0x10
CURSORRIGHT         = 0x14
LOADSYMBOL          = 0x40
SETCURSOR           = 0x80

#Line Addresses.
LINE = [0x00,0x40,0x14,0x54]    #for 20x4 display

battery = [
[ 0x0E, 0x1B, 0x11, 0x11, 0x11, 0x11, 0x11, 0x1F ],  #0%   (no charge)
[ 0x0E, 0x1B, 0x11, 0x11, 0x11, 0x11, 0x1F, 0x1F ],  #17%
[ 0x0E, 0x1B, 0x11, 0x11, 0x11, 0x1F, 0x1F, 0x1F ],  #34%
[ 0x0E, 0x1B, 0x11, 0x11, 0x1F, 0x1F, 0x1F, 0x1F ],  #50%  (half-full)
[ 0x0E, 0x1B, 0x11, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F ],  #67%
[ 0x0E, 0x1B, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F ],  #84%
[ 0x0E, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F ],  #100% (full charge)
]

patterns = [
[ 0x15, 0x0A, 0x15, 0x0A, 0x15, 0x0A, 0x15, 0x0A ],  #50%
[ 0x0A, 0x15, 0x0A, 0x15, 0x0A, 0x15, 0x0A, 0x15 ],  #alt 50%
[ 0x15, 0x15, 0x15, 0x15, 0x15, 0x15, 0x15, 0x15 ],  #3 vbars
[ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ],
[ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ],
[ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ],
[ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ],
```

```
]

verticalBars = [
[ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x1F ],  #1 bar
[ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x1F, 0x1F ],  #2 bars
[ 0x00, 0x00, 0x00, 0x00, 0x00, 0x1F, 0x1F, 0x1F ],  #3 bars
[ 0x00, 0x00, 0x00, 0x00, 0x1F, 0x1F, 0x1F, 0x1F ],  #4 bars
[ 0x00, 0x00, 0x00, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F ],  #5 bars
[ 0x00, 0x00, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F ],  #6 bars
[ 0x00, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F ],  #7 bars
[ 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F ],  #8 bars
]

horizontalBars = [
[ 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10 ],  #1 bar
[ 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18 ],  #2 bars
[ 0x1C, 0x1C, 0x1C, 0x1C, 0x1C, 0x1C, 0x1C, 0x1C ],  #3 bars
[ 0x1E, 0x1E, 0x1E, 0x1E, 0x1E, 0x1E, 0x1E, 0x1E ],  #4 bars
[ 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F ],  #5 bars
[ 0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F ],  #4 bars
[ 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07 ],  #3 bars
[ 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03 ],  #2 bars
]

digits = [
[ 0x01, 0x03, 0x03, 0x07, 0x07, 0x0F, 0x0F, 0x1F ],  #lower-rt triangle
[ 0x10, 0x18, 0x18, 0x1C, 0x1C, 0x1E, 0x1E, 0x1F ],  #lower-lf triangle
[ 0x1F, 0x0F, 0x0F, 0x07, 0x07, 0x03, 0x03, 0x01 ],  #upper-rt triangle
[ 0x1F, 0x1E, 0x1E, 0x1C, 0x1C, 0x18, 0x18, 0x10 ],  #upper-lf triangle
[ 0x00, 0x00, 0x00, 0x00, 0x1F, 0x1F, 0x1F, 0x1F ],  #lower horiz bar
[ 0x1F, 0x1F, 0x1F, 0x1F, 0x00, 0x00, 0x00, 0x00 ],  #upper horiz bar
[ 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F ]   #solid block
]

bigDigit = [
[ 0x00, 0x06, 0x01, 0x06, 0x20, 0x06, 0x06, 0x20, 0x06, 0x02, 0x06, 0x03], #0
[ 0x20, 0x06, 0x20, 0x20, 0x06, 0x20, 0x20, 0x06, 0x20, 0x20, 0x06, 0x20], #1
[ 0x00, 0x06, 0x01, 0x20, 0x00, 0x03, 0x00, 0x03, 0x20, 0x06, 0x06, 0x06], #2
[ 0x00, 0x06, 0x01, 0x20, 0x20, 0x06, 0x20, 0x05, 0x06, 0x02, 0x06, 0x03], #3
[ 0x06, 0x20, 0x06, 0x06, 0x06, 0x06, 0x20, 0x20, 0x06, 0x20, 0x20, 0x06], #4
[ 0x06, 0x06, 0x06, 0x06, 0x04, 0x04, 0x20, 0x20, 0x06, 0x06, 0x06, 0x03], #5
[ 0x00, 0x06, 0x01, 0x06, 0x20, 0x20, 0x06, 0x05, 0x01, 0x02, 0x06, 0x03], #6
[ 0x06, 0x06, 0x06, 0x20, 0x20, 0x06, 0x20, 0x20, 0x06, 0x20, 0x20, 0x06], #7
[ 0x00, 0x06, 0x01, 0x06, 0x20, 0x06, 0x06, 0x05, 0x06, 0x02, 0x06, 0x03], #8
[ 0x00, 0x06, 0x01, 0x02, 0x04, 0x06, 0x20, 0x20, 0x06, 0x20, 0x20, 0x06]  #9
]

########################################################################
#
#   Low-level routines for configuring the LCD module.
#      These routines contain GPIO read/write calls.
#

def InitIO():
       #Sets GPIO pins to input & output, as required by LCD board
    GPIO.setmode(GPIO.BCM)
    GPIO.setwarnings(False)
    for lcdLine in OUTPUTS:
        GPIO.setup(lcdLine, GPIO.OUT)
    for switch in INPUTS:
        GPIO.setup(switch, GPIO.IN, pull_up_down=GPIO.PUD_UP)

def CheckSwitches():
```

```
       #Check status of all four switches on the LCD board
       #Returns four boolean values as a tuple.
    val1 = not GPIO.input(SW1)
    val2 = not GPIO.input(SW2)
    val3 = not GPIO.input(SW3)
    val4 = not GPIO.input(SW4)
    return (val4,val1,val2,val3)

def PulseEnableLine():
    #Pulse the LCD Enable line; used for clocking in data
    GPIO.output(LCD_E, GPIO.HIGH)  #pulse E high
    GPIO.output(LCD_E, GPIO.LOW)   #return E low

def SendNibble(data):
    #sends upper 4 bits of data byte to LCD data pins D4-D7
    GPIO.output(LCD_D4, bool(data & 0x10))
    GPIO.output(LCD_D5, bool(data & 0x20))
    GPIO.output(LCD_D6, bool(data & 0x40))
    GPIO.output(LCD_D7, bool(data & 0x80))

def SendByte(data,charMode=False):
       #send one byte to LCD controller
    GPIO.output(LCD_RS,charMode)   #set mode: command vs. char
    SendNibble(data)               #send upper bits first
    PulseEnableLine()              #pulse the enable line
    data = (data & 0x0F)<< 4       #shift 4 bits to left
    SendNibble(data)               #send lower bits now
    PulseEnableLine()              #pulse the enable line


########################################################################
#
#   Higher-level routines for diplaying data on the LCD.
#

def ClearDisplay():
       #This command requires 1.5mS processing time, so delay is needed
       SendByte(CLEARDISPLAY)
       time.sleep(0.0015)           #delay for 1.5mS

def CursorOn():
       SendByte(CURSORON)

def CursorOff():
       SendByte(CURSOROFF)

def CursorBlink():
       SendByte(CURSORBLINK)

def CursorLeft():
       SendByte(CURSORLEFT)

def CursorRight():
       SendByte(CURSORRIGHT)

def InitLCD():
       #initialize the LCD controller & clear display
    SendByte(0x33)                 #initialize
    SendByte(0x32)                 #initialize/4-bit
    SendByte(0x28)                 #4-bit, 2 lines, 5x8 font
    SendByte(LEFTTORIGHT)          #rightward moving cursor
    CursorOff()
    ClearDisplay()
```

```
def SendChar(ch):
    SendByte(ord(ch),True)

def ShowMessage(string):
    #Send string of characters to display at current cursor position
    for character in string:
        SendChar(character)

def GotoLine(row):
    #Moves cursor to the given row
    #Expects row values 0-1 for 16x2 display; 0-3 for 20x4 display
    addr = LINE[row]
    SendByte(SETCURSOR+addr)

def GotoXY(row,col):
       #Moves cursor to the given row & column
       #Expects col values 0-19 and row values 0-3 for a 20x4 display
       addr = LINE[row] + col
       SendByte(SETCURSOR + addr)


########################################################################
#
#   BIG CLOCK & Custom character generation routines
#

def LoadCustomSymbol(addr,data):
       #saves custom character data at given char-gen address
       #data is a list of 8 bytes that specify the 5x8 character
       #each byte contains 5 column bits (b5,b4,..b0)
       #each byte corresponds to a horizontal row of the character
       #possible address values are 0-7
       cmd =  LOADSYMBOL + (addr<<3)
       SendByte(cmd)
       for byte in data:
              SendByte(byte,True)

def LoadSymbolBlock(data):
       #loads a list of symbols into the chargen RAM, starting at addr 0x00
       for i in range(len(data)):
              LoadCustomSymbol(i,data[i])

def ShowBigDigit(symbol,startCol):
       #displays a 4-row-high digit at specified column
       for row in range(4):
              GotoXY(row,startCol)
              for col in range(3):
                     index = row*3 + col
                     SendByte(symbol[index],True)

def ShowColon(col):
       #displays a 2-char high colon ':' at specified column
       dot = chr(0xA1)
       GotoXY(1,col)
       SendChar(dot)
       GotoXY(2,col)
       SendChar(dot)

def BigClock(seconds=10):
       #displays large-digit time in hh:mm:ss on 20x4 LCD
       #continuous display (this routine does not end!)
       print "  Big Clock running"
```

```
        LoadSymbolBlock(digits)
        posn = [0,3,7,10,14,17]
        ClearDisplay()
        ShowColon(6)
        ShowColon(13)
        for count in range(seconds):
                tStr = time.strftime("%I%M%S")
                for i in range(len(tStr)):
                        value = int(tStr[i])
                        symbols = bigDigit[value]
                        ShowBigDigit(symbols,posn[i])
                time.sleep(1)

########################################################################
#
#   Basic HD44780 Test Routines
#      Code here is used in higher-level testing routines
#

ANIMATIONDELAY = 0.02

def LabelTest(label):
        #Label the current Test
        ClearDisplay()
        GotoXY(1,20-len(label)); ShowMessage(label)
        GotoXY(2,16); ShowMessage('test')

def CommandTest():
        LabelTest('Command')
        while (True):
                st = raw_input("Enter a string or command: ")
                if len(st)==2:
                        SendByte(int(st,16))
                elif len(st)==1:
                        SendByte(int(st),True)
                else:
                        ShowMessage(st)

def AnimateCharTest(numCycles=8,delay=ANIMATIONDELAY):
        LabelTest('Animation')
        LoadSymbolBlock(battery)                #get all battery symbols
        GotoXY(1,6)                             #where to put battery
        for count in range(numCycles):
                for count in range(len(battery)):  #sequence thru all symbols
                        SendByte(count,True)       #display the symbol
                        CursorLeft()               #keep cursor on same char
                        time.sleep(delay)          #control animation speed
                time.sleep(1)                      #wait between cycles

def ShowBars(row,col,numBars):
        #displays a graph symbol at row,col position
        #numBars = number of horizontal (or vertical bars) in this symbol
        #expected values = 0 to 7 (vertical) or 0 to 4 (horizontal)
        GotoXY(row,col)
        if numBars==0:
                SendChar(' ')
        else:
                SendByte(numBars-1,True)


########################################################################
#
#   Alphanumberic Testing Routines
```

```
#

def UpdateCursor(count):
	WIDTH = 15
	if count==0:
		GotoLine(0)
	elif count==WIDTH:
		GotoLine(1)
	elif count==WIDTH*2:
		GotoLine(2)
	elif count==WIDTH*3:
		GotoLine(3)

def GetNextCharacter(code):
	#for a given CODE, returns the next displayable ASCII character
	#for example, calling with 'A' will return 'B'
	#removes nondisplayable characters in HD44780 character set
	if (code<0x20) or (code>=0xFF):
		code = 0x20
	elif (code>=0x7F) and (code<0xA0):
		code = 0xA0
	else:
		code += 1
	return code

def FillScreen(code,delay=0):
	#fill the LCD display with ASCII characters, starting with CODE,
	#assumes a width of 15 characters x 4 lines = 60 chars total
	for count in range(60):
		UpdateCursor(count)
		SendByte(code,True)
		code = GetNextCharacter(code)
		time.sleep(delay)

def FillChar(code):
	#fill the LCD display with a single ASCII character
	#assumes a width of 15 characters x 4 lines = 60 chars total
	for count in range(60):
		UpdateCursor(count)
		SendByte(code,True)

def CharTest(numCycles=4, delay=ANIMATIONDELAY):
	#show screenfull of sequential symbols from character set
	#starting with a random symbol
	#delay = time between characters
	LabelTest('Char')
	for count in range(numCycles):
		rand   = random.randint(0,255)
		firstChar = GetNextCharacter(rand)
		FillScreen(firstChar,delay)

def NumberTest(delay=1):
	#show an almost-full screen (60 chars) of each digit 0-9
	#call with delay in seconds between each digit/screen
	for count in range(10):
		FillChar(ord('0')+count)
		time.sleep(delay)

def TimeTest(numCycles=3):
	#measures the time required to display 600 characters, sent
	#60 characters at a time.  The pause between screen displays
	#is removed from the reported time.
	pause = 0.5
```

```
	LabelTest('Time')
	for count in range(numCycles):
		startTime = time.time()
		NumberTest(pause)
		elapsedTime = time.time()-startTime
		elapsedTime -= pause*10
		print "  elapsed time (sec): %.3f" % elapsedTime


########################################################################
#
#   Horizontal Graph Testing Routines
#

def ClearHBar(row,startCol):
	#remove all elements on horizontal bar
	GotoXY(row,startCol)
	for col in range(12):
		SendByte(0x20,True)

def HBar(length,row):
	#creates a horizontal bar on the specified row
	#expects length of 1 (min) to 80 (max)
	#Must load horizontal bar symbols prior to calling
	fullChars = length / 5
	bars      = length % 5
	col       = 3
	ClearHBar(row,col)
	for count in range(fullChars):
		ShowBars(row,col,5)
		col += 1
	if bars>0:
		ShowBars(row,col,bars)

def HBarTest(numCycles=8):
	LoadSymbolBlock(horizontalBars)
	LabelTest('Horz')
	for count in range(numCycles):
		for row in range(4):
			length = random.randint(1,60)
			GotoXY(row,0)
			ShowMessage("%2d" % length)
			HBar(length,row)
		time.sleep(1)

def DecrementHBar(row,startCol,length):
	#reduce the number of horizontal bars by one
	length -= 1
	fullChars = length / 5
	bars      = length % 5
	col = startCol + fullChars
	ShowBars(row,col,bars)

def IncrementHBar(row,startCol,length):
	#increase the number of horizontal bars by one
	fullChars = length / 5
	bars      = length % 5
	col = startCol + fullChars
	ShowBars(row,col,bars+1)

def AnimatedHBar(row,startCol,newLength,oldLength=0):
	diff = newLength - oldLength
	for count in range(abs(diff)):
```

```
            if diff>0:
                IncrementHBar(row,startCol,oldLength)
                oldLength +=1
            else:
                DecrementHBar(row,startCol,oldLength)
                oldLength -=1
            time.sleep(ANIMATIONDELAY)

def AnimatedHBarTest(numCycles=8):
    LoadSymbolBlock(horizontalBars)
    LabelTest('HBar')
    graph = [0,0,0,0]
    for count in range(numCycles):
        for row in range(4):
            length = random.randint(1,60)
            GotoXY(row,0)
            ShowMessage("%2d" % length)
            AnimatedHBar(row,3,length,graph[row])
            graph[row] = length


########################################################################
#
#   Vertical Graph Testing Routines
#

def ClearVBar(col):
    #remove all elements on a vertical bar
    for row in range(4):
        GotoXY(row,col)
        SendByte(0x20,True)

def VBar(height,col):
    #creates a vertical bar at specified column
    #expects height of 1 (min) to 32 (max)
    #Must load vertical bar symbols prior to calling
    fullChars = height / 8
    bars      = height % 8
    row = 3
    ClearVBar(col)
    for count in range(fullChars):
        ShowBars(row,col,8)
        row -= 1
    if bars>0:
        ShowBars(row,col,bars)

def VBarTest(numCycles=4):
    LoadSymbolBlock(verticalBars)
    LabelTest('Vert')
    for count in range(numCycles):
        for col in range(15):
            height = random.randint(1,32)
            VBar(height,col)
        time.sleep(1)

def SineGraph(numCycles=4):
    #print a sin wave function using vertical bars.
    #this is a sample application of the VBar routine.
    #the 'sine' list emulates the following formula:
    #radians=x*2*math.pi/15; y = math.sin(radians)*15 + 16
    sine = [16,22,27,30,31,29,25,19,13,7,3,1,2,5,10]
    LoadSymbolBlock(verticalBars)
    LabelTest('Sine')
```

```
	for count in range(numCycles):
		for step in range(15):
			for col in range(15):
				x = col+step
				VBar(sine[x%15],col)
			time.sleep(0.2)

def IncrementVBar(col,height):
	#increaase the number of vertical bars by one
	fullChars = height / 8
	bars      = height % 8
	ShowBars(3-fullChars,col,bars+1)

def DecrementVBar(col,height):
	#decrease the number of vertical bars by one
	height -= 1
	fullChars = height / 8
	bars      = height % 8
	ShowBars(3-fullChars,col,bars)

def AnimatedVBar(col,newHeight,oldHeight=0):
	diff = newHeight - oldHeight
	for count in range(abs(diff)):
		if diff>0:
			IncrementVBar(col,oldHeight)
			oldHeight +=1
		else:
			DecrementVBar(col,oldHeight)
			oldHeight -=1
		time.sleep(ANIMATIONDELAY)

def AnimatedVBarTest(numCycles=4):
	LoadSymbolBlock(verticalBars)
	LabelTest('VBar')
	graph = [0]*15
	for count in range(numCycles):
		for col in range(15):
			height = random.randint(1,32)
			AnimatedVBar(col,height,graph[col])
			graph[col] = height


########################################################################
#
#   Main Program
#

print "Pi LCD4 program starting."
InitIO()                            #Initialization
InitLCD()
ClearDisplay()
CharTest()
TimeTest()                          #Basic LCD Tests
AnimateCharTest()
HBarTest()                          #Horizontal Graph Tests
AnimatedHBarTest()
VBarTest()                          #Vertical Graph Tests
AnimatedVBarTest()
SineGraph()
BigClock()                          #Something actually useful

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