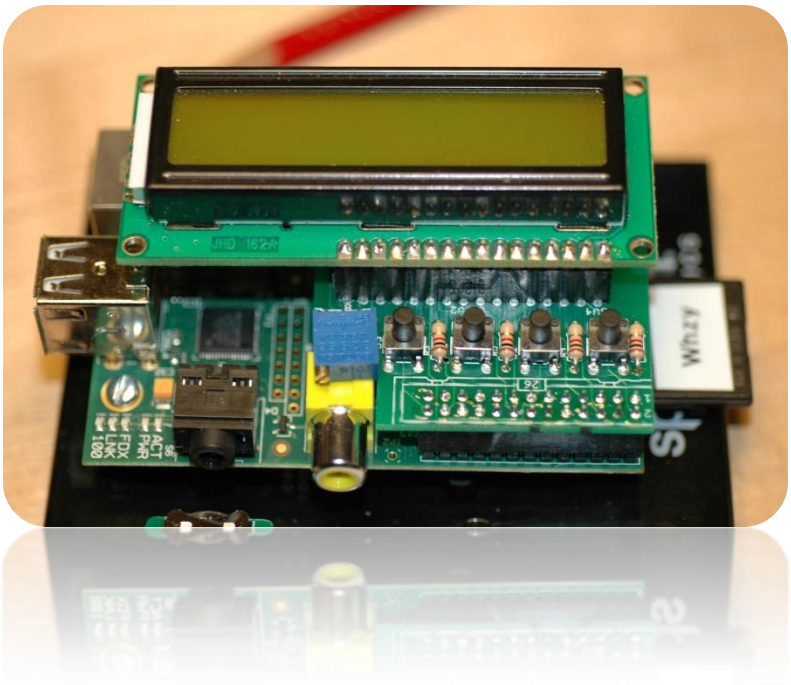


Beginner's Guide to the PI LCD

Part 2: Display

, W8BH

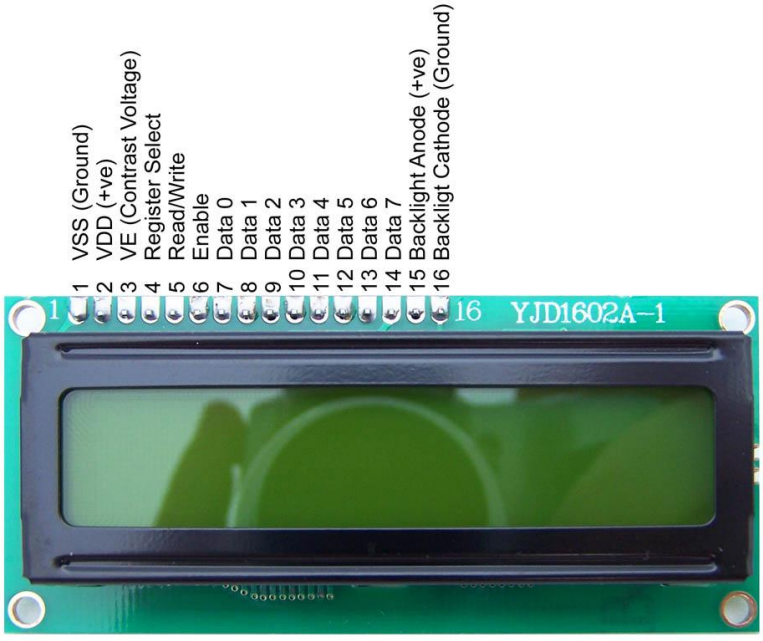
1) INTRODUCTION

In Part 1 of this series, we assembled the Pi LCD from mypishop.com, and tested the 'wiring and the switches'. Now it's time to concentrate on the display itself. You may have purchased either a 16x2 or a 20x4 display with your kit. I'll assume that you have the 16x2 display, and show you the small changes required for using the larger display.

2) THE HD44780 LCD CONTROLLER

Our display, like many backlit LCD modules available today, uses a 16-pin interface. We should thank Hitachi for developing this widely used, informal standard.

Pins 1 & 2 power the controller. Pin 3 is connected to the potentiometer and used to adjust display contrast. Pins 15 & 16 power the backlight. On our board, we don't need to read data from the display, so we keep the Read/Write line tied to ground. This leaves 10 digital pins we can use for data communication.



The LCD controller gives us two ways to send data: in one-byte (8 bit) chunks, or as two consecutive nibbles (4 bit). The second approach seems more complicated, but very frequently used. Why? It simplifies the hardware, and lets us talk to the LCD with half as many data lines. In general, I/O lines are a scarce resource! We use the 4-bit method on this board, and need only 6 I/O lines (2 control + 4 data) to communicate with the LCD.

Here are the digital I/O connections between our Pi and the LCD controller:

GPIO#	LCD Pin	Function
7	4	Register Select
8	6	Enable
17	11	Bit 0 (Data line D4)
18	12	Bit 1 (Data line D5)
27	13	Bit 2 (Data line D6)
22	14	Bit 3 (Data line D7)

Now we have enough information to program our data lines on the Pi. We need to set up 6 GPIO lines as outputs. I like to give each line a name, rather than just a number

```
GPIO.setup(LCD_D4,GPIO.OUT)
GPIO.setup(LCD_D5,GPIO.OUT)
GPIO.setup(LCD_D6,GPIO.OUT)
…
```

If you want to, you can put all the output lines in a list, and then loop through them:

```
OUTPUTS = [LCD_RS, LCD_E, LCD_D4, LCD_D5, LCD_D6, LCD_D7]
for ioLine in OUTPUTS:
      GPIO.setup(ioLine,GPIO.OUT)
```

3) NIBBLES & BYTES

Put on your thinking cap, because it's time for the hard part: sending data to the LCD controller. There are 8 bits to each byte, but we can only send 4 bits at a time. And we have to time them according to the controller's specifications. Check out the datasheet for the specific details. The gist is to send the upper 4 bits of the data, toggle the enable pin, and then send the lower 4 bits. The half-byte chunks are called nibbles.

```
def SendByte (data):
      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

def PulseEnableLine():
      #Pulse the LCD Enable line; used for clocking in data
      mSec = 0.0005                 #use half-millisecond delay
      time.sleep(mSec)              #give time for inputs to settle
      GPIO.output(LCD_E, GPIO.HIGH) #pulse E high
```

```
        time.sleep(mSec)
        GPIO.output(LCD_E, GPIO.LOW)  #return E low
        time.sleep(mSec)              #wait before doing anything else
```

Each waiting period is specified in the HD44780 datasheet. You can minimize wait times by following the specifications exactly. I chose a lazier method. I empirically picked a delay that is longer than necessary, but short enough to minimize visual distractions: one millisecond. It works. Even half a millisecond is OK on my displays. Go much shorter, however, and you'll need to account for all of the timing requirements listed in the spec. It is up to you.

Bytes can be send to the controller as either data byte (characters) or as commands. The controller uses the input line RS to distinguish the two: anything sent when RS is low is a command, and anything sent when RS is high is a character. We'll modify our SendByte routine to account for this requirement.

```
def SendByte(data,charMode=False):
        GPIO.output(LCD_RS,charMode)  #set mode: command vs. char
        SendNibble(data)              #send upper bits first
        …etc…
```

By using a default parameter, any call to SendByte will default to a command. Let's create a second routine for sending characters:

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

Now we have routines for sending commands and characters to the display. It would be nice to test them right away, but we can't: we have to initialize the display first. All HD44780-based displays require certain startup commands to specify things like data-length, cursor-mode, etc. Without going into a lot of detail here, our display requires the following "magic bytes" to initialize it: 0x33, 0x32, 0x28, 0x0C, 0x06, 0x01. These command bytes will set the data-length to 4 bits, turn the cursor off, enable sequential addressing, and clear the display.

```
def InitLCD():
        SendByte(0x33)              #initialize
        SendByte(0x32)              #set to 4-bit mode
        SendByte(0x28)              #2 line, 5x7 matrix
        …etc…
```

It's time to write something on the LCD display. Write a routine to display a string, like 'Hello, World'. All we need to do is write each character, one at a time. A simple for-loop will do the trick:

```
def ShowMessage(string):
        for character in string:
                SendChar(character)
```

4) INPUT & OUTPUT

Let's combine our string-writing ability with the switch input routine from part 1. In Part 1 we read the status of each switch and displayed it on the console. Now we will display switch status on the LCD screen instead. First, write something meaningful on the display:

WriteMessage('Press a switch...')

And put the switch status on the second display line. Wait a sec, how do we put stuff on the second line? Send a carriage return? Sorry, that doesn't work. How about sending a full line of 16 characters to the display? Surely the next character will go on the next line. No, sorry again.

Characters sent to the LCD controller are placed at the current cursor position. In sequential mode, the cursor is advanced with each character sent. But unfortunately, second-line addresses do not immediately follow the first line. The cursor address for Line1 is 0x00. The cursor address for Line 2 is 0x40. Strange, but true. If you have a 20x4 display, the arrangement is even more confusing:

20x4 Display	Address
Line 1	0x00
Line 2	0x40
Line 3	0x14
Line 4	0x54

16x2 Display	Address
Line 1	0x00
Line 2	0x40

To set the cursor position, send a byte equal to the set cursor command (0x80) + the desired cursor address. Now we can put together our demo. Four true/false results do not all fit on a 16 character line. Using the string format of "%d %d %d %d", the boolean results are converted to more compact (d for decimal) ones and zeros.

```
while (True):
        GotoLine(1)
        switchValues = CheckSwitches()
        decimalResult = "  %d  %d  %d  %d" % switchValues
        ShowMessage(decimalResult)
        time.sleep(0.2)
```

That's it for part 2. We can now read the status of each switch and display short messages on the LCD display. In part 3 we will add useful display routines for cursor control, text positioning, and scrolling.

6) PYTHON SCRIPT for PI LCD, PART 2:

```
#############################################################################
#
#    LCD2:  Learning how to control an LCD module from Pi
#
#



#
#############################################################################

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

#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 (Use 21 on Rev.1 Pi's)
LCD_D7              = 22        #GPIO22 = Pi pin 15
OUTPUTS = [LCD_RS,LCD_E,LCD_D4,LCD_D5,LCD_D6,LCD_D7]

#INPUTS: map GPIO to Switches
SW1                 = 4         #GPIO4  = Pi pin 7
SW2                 = 23        #GPIO16 = Pi pin 16
SW3                 = 10        #GPIO10 = Pi pin 19
SW4                 = 9         #GPIO9  = Pi pin 21
INPUTS = [SW1,SW2,SW3,SW4]


#HD44780 Controller Commands
CLEARDISPLAY        = 0x01
SETCURSOR           = 0x80

#Line Addresses.  (Pick one.  Comment out whichever doesn't apply)
#LINE = [0x00,0x40,0x14,0x54]    #for 20x4 display
LINE = [0x00,0x40]              #for 16x2 display


#############################################################################
#
#   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
    mSec = 0.0005                   #use half-millisecond delay
    time.sleep(mSec)                #give time for inputs to settle
    GPIO.output(LCD_E, GPIO.HIGH)   #pulse E high
    time.sleep(mSec)
    GPIO.output(LCD_E, GPIO.LOW)    #return E low
    time.sleep(mSec)                #wait before doing anything else

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

def InitLCD():
    #initialize the LCD controller & clear display
    SendByte(0x33)              #initialize
    SendByte(0x32)              #set to 4-bit mode
    SendByte(0x28)              #2 line, 5x7 matrix
    SendByte(0x0C)              #turn cursor off (0x0E to enable)
    SendByte(0x06)              #shift cursor right
    SendByte(CLEARDISPLAY)      #remove any stray characters on display


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

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)
```

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

print "Pi LCD2 program starting.  Ctrl-C to stop."
InitIO()
InitLCD()
ShowMessage('Press a switch!')
while (True):
    GotoLine(1)
    switchValues = CheckSwitches()
    decimalResult = "  %d  %d  %d  %d" % switchValues
    ShowMessage(decimalResult)
    time.sleep(0.2)

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