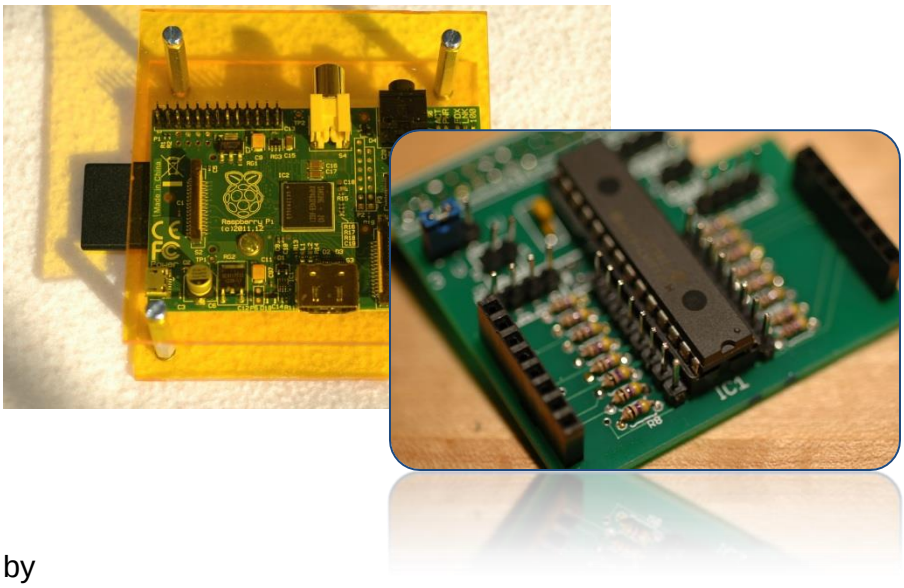


Beginner's Guide to the PI MATRIX

Part 3 :
The Toolkit

by
Bruce E. Hall, W8BH

1) INTRODUCTION

In Part 1 and Part 2 of this series, we looked at how to build this great little kit, the Pi Matrix, and how to program simple display routines in Python. In this tutorial we'll develop those routines a bit further, giving us a whole toolkit of interesting displays to choose from.

2) COMMERCIALWARE vs. HOBBYWARE

This is an educational hobby for me, and has been a great way to learn about the Raspberry Pi and about Python programming. But I am not an expert Python programmer by any means. My coding style – if I have a coding style – comes from other languages and is not very 'Pythonic'. My use of whitespace is inconsistent. My procedure and variable naming is inconsistent. But I enjoy learning by doing, and I hope you do, too!

3) SIMPLIFY, SIMPLIFY

There are all sorts of ways to encapsulate data and code. We break it up into files. We mold it into structures and functions. A great object-oriented way to think about this project is to create a 'matrix' class. This object would include data & functions relevant to the raspberry pi, and insulate the user from the messy lower level details. Great idea – go for it! I decided to take a simpler approach here, and just make some useful functions that anyone can grab and add to a project.

Let's start with some low-level routines that write data to the MCP27013 chip. In Part 2 we made a column-writing routine like this: write a routine to turn on a single column. To turn on the first column, we just need to turn on the lowest bit. This is easy: just write the value of '00000001' (0x01) to Port A like this:

```
def SetColumn (col):
        bus.write_byte_data(ADDR,PORTB,0x00)
        bus.write_byte_data(ADDR,PORTA,0x80>>column)
```

All of the routines that write to the chip will look like this, writing byte values to one of more chip registers. They work fine as is. I decided to simplify them just a bit, taking out the repetitive parts like this:

```
def Write (register, value):
        bus.write_byte_data(ADDR, register, value)

def SetColumn (col):
        Write (PORTB, 0x00)
        Write (PORTA, 0x80>>column)
```

We don't need to do this, but it moves all of the I2C calls to a single routine and removes the possibility of sending some of our data to the wrong I2C bus address. The simplest routine is one that writes data to the LED column and row inputs. Once we write that, we can further simplify SetColumn like this:

```
def WriteToLED (rowPins, colPins):
        Write (PORTA, colPins)
        Write (PORTB, rowPins)

def SetColumn (col):
        WriteToLED (0x00, 0x80>>column)
```

Again, this abstraction is not necessary but helpful: once WriteToLED is defined, we no longer have to remember whether columns go on PORTA or PORTB. Try it now and see!

4) PATTERNS

We still have to remember that rows are active LOW, and that the column bits are reversed. Wouldn't it be nice to call a function with first row bit and first column bit set, and have the top/left LED light up? It's easy! We'll use what we know, and then never have to remember it again:

	C0	C1	C2	C3	C4	C5	C6	C7
R0								
R1		X	X	X				
R2		X	X	X				
R3								
R4								
R5								
R6								
R7								

```
def SetPattern (rows, columns):
        WriteToLED(~rows, ReverseBits(columns))
```

The tilde(~) negates to row bits, making them all active LOW. And we'll need to write a routine to reverse the order of the column bits. But that's all it takes. For example, suppose

we want to make a small 2x2 square as shown above. We need to set row bits 1 &2 (0b00000110 = 0x02), and column bits 1, 2, and 3. (0b00001110 = 0x0E). Try calling SetPattern with some different bit patterns on the rows and columns to see what you can make.

5) ORIENTATION

The cables around my Pi hold it in an upside-down orientation. By that, I mean that the USB ports are on the left, the power is on the right, and the GPIO pins are facing me on the bottom. I was constantly turning my head around, trying to see if the correct LEDs were lit. Either I could turn the whole contraption around (not so easy), or I could rewrite my code so that it displayed upside down. Necessity is the mother of invention, so what would it take to flip the display?

Things get a little confusing here, I admit. But to flip a display, up means down and left means right. What? Something in the top-left, when flipped, is now bottom-right. OK, let's try reversing the row and column bits, and see if it works:

```
def SetPattern180 (rows,columns):
    SetPattern(ReverseBits(rows), ReverseBits(columns))
```

Hey, it works! We are on a roll. While we are at it, can we also change the orientation sideways left and right (90 and 270 degrees)? Sure thing: swap rows and columns, and reverse one of them. All of this reversing gets a bit cumbersome, since we are already reversing the columns in our original SetPattern routine. One solution combines both operations in a new SetPattern, like this:

```
def SetPattern (rows,columns,orientation=0):
    if orientation==0:
        WriteToLED(~rows,ReverseBits(columns))
    elif orientation==90:
        WriteToLED(~columns,rows)
    elif orientation==180:
        WriteToLED(~ReverseBits(rows),columns)
    elif orientation==270:
        WriteToLED(~ReverseBits(columns),ReverseBits(rows))
```

Our routine bulked up a bit, but now we can use the Pi at any angle. Notice the default value of 0 in the definition: we don't even need to specify any orientation at all, if the Pi happens to be right-side up. Call this routine with something other than the four values, and it's not going to work. It's hobby-ware!

6) BACK TO BASICS

Now that we can create any pattern, in positive logic, in any orientation, it is much easier to set an individual LEDs, columns, and rows.

```
def SetLED (row,col):
        SetPattern(1<<row, 1<<col)

def SetColumn (row,col):
        SetPattern(0xFF, 1<<col)

def SetRow (row,col):
        SetPattern(1<<row, 0xFF)
```

On my upside-down board, I set the orientation in SetPattern to 180 instead of 0.

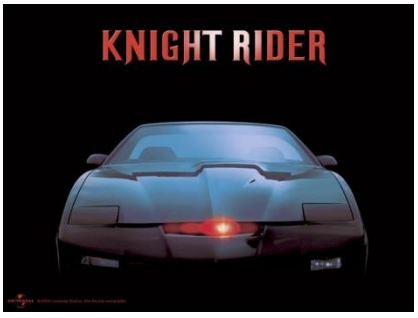
7) PLAY TIME!

In the last part we created some test patterns to exercise our LED matrix. Here are a few more. One of my favorites is the 'cylon'. The Cylon, a creature from the TV series Battlestar Galactica, had a red eye that shifted position back and forth across its face. Some of you may also remember KITT in Knight Rider that used the same effect.

```
def MultiRowCylon (pattern,numCycles):
        for count in range(0, numCycles):
                for col in range(0,7):
                        SetPattern(pattern, 1<<col)
                        time.sleep(delay)
                for col in range(7,0,-1):
                        SetPattern(pattern, 1<<col)
                        time.sleep(delay)

def SingleRowCylon (row,numCycles):
        #Side-to-side LED chaser, single row
        MultiRowCylon(1<<row, numCycles)

def AllRowCylon (numCycles):
        #Side-to-side LED chaser using all rows
        MultiRowCylon(0xFF, numCycles)
```



If you just have a row of LEDs, you can only do a single row effect. But we have the mighty Pi Matrix, so we can do cylon all the rows, or any combination in between. There isn't much to the code: just pick a row pattern, and then oscillate the columns back and forth.

You can do really fun animations just by experimenting and trying different patterns. Check out the YouTube video I created at http://youtu.be/VbPBNmlGy34

8) PYTHON SCRIPT for PART3:

```
#!/usr/bin/python

################################################################################
#
#    Matrix3:  Library of Pi Matrix display routines
#
#    This Python script contains a bunch of routines for driving
#    the Pi Matrix & creating interesting displays.
#    Online References: w8bh.net & mypishop.com
#    YouTube video at http://youtu.be/VbPBNmlGy34
#
#    Author:  Bruce E. Hall <bhall66@gmail.com>
#    Date  :  20 Feb 2013
#
################################################################################

import smbus             #gives us a connection to the I2C bus
import time              #for timing delays
import random            #random number generator

#Definitions for the MCP23017 chip
ADDR   = 0x20            #The I2C address of our chip
DIRA   = 0x00            #PortA I/O direction, by pin. 0=output, 1=input
DIRB   = 0x01            #PortB I/O direction, by pin. 0=output, 1=input
PORTA  = 0x12            #Register address for PortA
PORTB  = 0x13            #Register address for PortB


ORIENTATION = 180        #default viewing angle for the pi & matrix
                         #set this value according to your viewing angle
                         #values are  0 (power jack on left)
                         #           90 (power jack on bottom)
                         #          180 (power jack on right)
                         #          270 (power jack on top)

rows   = 0x00            #starting pattern is (0,0) = all LEDS off
columns= 0x00
delay  = 0.08            #time delay between LED display transitions
                         #smaller values = faster animation

################################################################################
#
#   Lower level routines for bit-manipulation.
#   Nothing here relates to our snazzy Pi Matrix.
#

def ReverseBits (byte):
    #reverse the bit order in the byte: bit0 <->bit 7, bit1 <-> bit6, etc.
    value = 0
    currentBit = 7
    for i in range(0,8):
        if byte & (1<<i):
            value |= (0x80>>i)
            currentBit -= 1
    return value
```

```
def ROR (byte):
    #perform a 'rotate right' command on byte
    #bit 1 is rotated into bit 7; everything else shifted right
    bit1 = byte & 0x01          #get right-most bit
    byte >>= 1                  #shift right 1 bit
    if bit1:                    #was right-most bit a 1?
        byte |= 0x80            #if so, rotate it into bit 7
    return byte

def ROL (byte):
    #perform a 'rotate left' command on byte
    #bit 7 is rotated into bit 1; everything else shifted left
    bit7 = byte & 0x080        #get bit7
    byte <<= 1                 #shift left 1 bit
    byte &= 0xFF               #only keep 8 bits
    if bit7:                   #was bit7 a 1?
        byte |= 0x01           #if so, rotate it into bit 1
    return byte


#########################################################################
#
#   Lower Level LED Display routines.
#   These write data directly to the Pi Matrix board
#

def Write (register, value):
    #Abstraction of I2C bus and the MCP23017 chip
    #Call with the chip data register & value pair
    #The chip address is constant ADDR (0x20).
    bus.write_byte_data(ADDR,register,value)

def EnableLEDS ():
    #Set up the 23017 for 16 output pins
    Write(DIRA, 0x00)           #all zeros = all outputs on PortA
    Write(DIRB, 0x00)           #all zeros = all outputs on PortB


def DisableLEDS ():
    #Set all outputs to high-impedance by making them inputs
    Write(DIRA, 0xFF);          #all ones = all inputs on PortA
    Write(DIRB, 0xFF);          #all ones = all inputs on PortB

def TurnOffLEDS ():
    #Clear the matrix display
    Write(PORTA, 0x00)          #set all columns low
    Write(PORTB, 0x00)          #set all rows low

def TurnOnAllLEDS ():
    #Turn on all 64 LEDs
    Write(PORTA, 0xFF)          #set all columns high
    Write(PORTB, 0x00)          #set all rows low

def WriteToLED (rowPins,colPins):
    #set logic state of LED matrix pins
    Write(PORTA, 0x00)          #turn off all columns; prevent ghosting
```

```
    Write(PORTB, rowPins)       #set rows first
    Write(PORTA, colPins)       #now turn on columns


#########################################################################
#
#   More low-level routines, saved for learning purposes.
#
#

def FastSetLED (row,col):
    #turn on an individual LED at (row,col).  All other LEDS off.
    #ignores orientation.  Unless speed is required, use SetLED instead
    bus.write_byte_data(ADDR,PORTA,0x80>>col)
    bus.write_byte_data(ADDR,PORTB,~(1<<row))

def FastSetColumn (col):
    #turn on all LEDs in the specified column. Expects input of 0-7.
    #ignores orientation.  Unless speed is required, use SetColumn.
    bus.write_byte_data(ADDR,PORTB,0x00)
    bus.write_byte_data(ADDR,PORTA,0x80>>col)

def FastSetRow (row):
    #turn on all LEDs in the specified row.  Expects input of 0-7.
    #ignores orientation.  Unless speed is required, use SetRow.
    bus.write_byte_data(ADDR,PORTA,0xFF)
    bus.write_byte_data(ADDR,PORTB,~(1<<row))


#########################################################################
#
#   Intermediate-level routines for
#   LED Pixel, Row, Column, and Pattern display.
#   Set constant ORIENTATION to 0,90,180,270 according to Matrix posn.
#

def SetPattern (rowPattern,colPattern,orientation=ORIENTATION):
    #Applies given row & column patterns to the matrix.
    #For columns, bit 0 is left-most and bit 7 is at far right.
    #For rows, bit 0 is at the top and bit 7 is at the bottom.
    #Example: (0x07,0x03) will set 3 row bits & 2 columns bits,
    #forming a rectagle of 6 lit LEDS in upper left corner of
    #the matrix, three rows tall and two columns wide.
    #Why?  0x07 = 0b00000111 (three lower row bits set).
    #      0x03 = 0b00000011 (two lower column bits set).


    global rows, columns       #save current row/column
    rows = rowPattern
    columns = colPattern

    if orientation==0:
        WriteToLED(~rows,ReverseBits(columns))
    elif orientation==90:
        WriteToLED(~columns,rows)
    elif orientation==180:
        WriteToLED(~ReverseBits(rows),columns)
```

```
    elif orientation==270:
        WriteToLED(~ReverseBits(columns),ReverseBits(rows))


def SetLED(row,col):
    #turn on an individual LED at (row,col).  All other LEDS off.
    #expects inputs of (0,0) to (7,7).
    SetPattern(1<<row,1<<col)

def SetColumn(col):
    #turn on all LEDs in the specified column. Expects input of 0-7.
    SetPattern(0xFF,1<<col)

def SetRow(row):
    #turn on all LEDs in the specified row. Expects input of 0-7.
    SetPattern(1<<row,0xFF)

def MoveDown():
    #shifts entire display downward by one pixel
    SetPattern(rows<<1,columns)

def MoveUp():
    #shifts entire display downward by one pixel
    SetPattern(rows>>1,columns)

def MoveRight():
    #shifts entire display one pixel to the right
    SetPattern(rows,columns<<1)

def MoveLeft():
    #shifts entire display one pixel to the left
    SetPattern(rows,columns>>1)


#########################################################################
#
#   Higher-level routines for playing with the Pi Matrix board.
#   Each routine does a simple display animation, calling
#   one or more of the low-level routines above.
#   Set global variable DELAY to change animation speed (default 0.08)
#

def ShowRoutine (name):
    #display the routine name to console, file, or wherever you want.
    #this is useful for creating a log of your display.
    print " ",name

def FlashLEDS (delay):
    #Flash all of the LEDS on/off for the specified time
    TurnOnAllLEDS()
    time.sleep(delay)
    TurnOffLEDS()
    time.sleep(delay)

def MultiRowCylon (pattern=0x55,numCycles=4):
    #Do a side-to-side LED chaser using multiple rows
    ShowRoutine("%d Row Cylon %x" % (numCycles,pattern))
```

```
    for count in range(0,numCycles):
        for col in range(0,8):
            SetPattern(pattern,1<<col)
            time.sleep(delay)
        for col in range(6,0,-1):
            SetPattern(pattern,1<<col)
            time.sleep(delay)

def SingleRowCylon (row=0,numCycles=4):
    #Do a side-to-side LED chaser on a single row
    MultiRowCylon(1<<row,numCycles)

def AllRowCylon (numCycles=4):
    #Do a side-to-side LED chaser using all rows
    MultiRowCylon(0xFF,numCycles)

def MultiColumnCylon (pattern=0x55,numCycles=4):
    #Do a back-and-forth LED chaser on multiple columns
    ShowRoutine("%d Column cylon %x" % (numCycles,pattern))
    for count in range(0,numCycles):
        for row in range(0,8):
            SetPattern(1<<row,pattern)
            time.sleep(delay)
        for row in range(6,0,-1):
            SetPattern(1<<row,pattern)
            time.sleep(delay)

def MultiBarCylon (pattern=0x77,numCycles=4):
    #Do a back-and-forth LED chaser on multiple columns
    ShowRoutine("%d Bar cylon %x" % (numCycles,pattern))
    for count in range(0,numCycles):
        for row in range(0,8):
            SetPattern(~1<<row,pattern)
            time.sleep(delay)
        for row in range(6,0,-1):
            SetPattern(~1<<row,pattern)
            time.sleep(delay)

def SingleColumnCylon (col=0,numCycles=4):
    #Do an up-and-down LED chaser on a single column
    MultiColumnCylon(1<<col,numCycles)

def AllColumnCylon (numCycles=4):
    #Do an up-and-down LED chaser using all columns
    MultiColumnCylon(0xFF,numCycles)

def Marquis (indent=0,numCycles=8,delay=0.02):
    #do a LED chaser around perimeter of matrix
    ShowRoutine("%d Marquis #%d at %g" % (numCycles,indent,delay))
    for count in range(0,numCycles):
        for col in range(0+indent,8-indent):
            SetLED(0+indent,col)
            time.sleep(delay)
        for row in range(1+indent,7-indent):
            SetLED(row,7-indent)
            time.sleep(delay)
        for col in range(7-indent,-1+indent,-1):
```

```
            SetLED(7-indent,col)
            time.sleep(delay)
        for row in range(6-indent,0+indent,-1):
            SetLED(row,0+indent)
            time.sleep(delay)

def RotateDisplay (direction, numCycles=32):
    #rotates current display in desired direction
    global rows,columns
    ShowRoutine("%d Rotate display %s" %(numCycles,direction))
    time.sleep(delay)                       #initial pattern = first cycle
    for count in range(0,numCycles):
        if direction=="down":
            rows = ROL(rows)
        elif direction=="up":
            rows = ROR(rows)
        elif direction=="right":
            columns = ROL(columns)
        elif direction=="left":
            columns = ROR(columns)
        SetPattern(rows,columns)
        time.sleep(delay)

def RandomPatterns (numCycles=32):
    #puts random display patterns on the LED matrix
    ShowRoutine("%d Random Patterns" % numCycles)
    delay = 0.05
    for count in range(0,numCycles):
        rowPattern = random.randint(0,255)
        colPattern = random.randint(0,255)
        SetPattern(rowPattern,colPattern)
        time.sleep(delay)

def RandomPixels (numCycles=128):
    #puts random display patterns on the LED matrix
    ShowRoutine("%d Random Pixels" % numCycles)
    delay = 0.02
    for count in range(0,numCycles):
        row = random.randint(0,7)
        col = random.randint(0,7)
        SetLED (row,col)
        time.sleep(delay)

def ShowEyes ():
    SetPattern(0x18,0x66)

def BlinkEyes (numCycles=8,delay=0.5):
    for count in range(0,numCycles):
        ShowEyes()
        time.sleep(delay)
        TurnOffLEDS()
        time.sleep(delay *0.2)

def Blink (numCycles=8,delay=0.5):
    #flashes current display off, then back on
    for count in range(0,numCycles):
        time.sleep(delay)
```

```
        DisableLEDS()
        time.sleep(delay)
        EnableLEDS()

def Figure8 (numCycles=4,delay=0.05):
    ShowRoutine("%d Figure8" % numCycles)
    SetPattern(3,3)
    time.sleep(delay*3)
    for cycle in range(0,numCycles):
        for count in range(0,6):
            MoveRight()
            time.sleep(delay)
        for count in range(0,6):
            MoveDown()
            MoveLeft()
            time.sleep(delay)
        for count in range(0,6):
            MoveRight()
            time.sleep(delay)
        for count in range(0,6):
            MoveUp()
            MoveLeft()
            time.sleep(delay)

def PawCircle(delay=0.5):
    ShowRoutine("Paw Circle")
    SetPattern(0x33,0x03)
    time.sleep(delay)
    for count in range(0,3):
        MoveRight()
        MoveRight()
        time.sleep(delay)
    SetPattern(0xCC,0xC0)
    time.sleep(delay)
    for count in range(0,3):
        MoveLeft()
        MoveLeft()
        time.sleep(delay)

def Pulse1(numCycles=8,delay=0.50):
    #pulsing animation in corner of display
    ShowRoutine("%d Pulse1" % numCycles)
    for count in range(0,numCycles):
        SetPattern(0x80,0x01)
        time.sleep(0.03)
        SetPattern(0xC0,0x03)
        time.sleep(0.03)
        SetPattern(0xE0,0x07)
        time.sleep(0.03)
        SetPattern(0xC0,0x03)
        time.sleep(0.03)
        SetPattern(0x80,0x01)
        time.sleep(delay-0.12)

def Pulse4(numCycles=8,delay=0.50):
    #pulsing animation in all four display corners
    ShowRoutine("%d Pulse4" % numCycles)
```

```
    for count in range(0,numCycles):
        SetPattern(0x81,0x81)
        time.sleep(0.03)
        SetPattern(0xC3,0xC3)
        time.sleep(0.03)
        SetPattern(0xE7,0xE7)
        time.sleep(0.03)
        SetPattern(0xC3,0xC3)
        time.sleep(0.03)
        SetPattern(0x81,0x81)
        time.sleep(delay-0.12)

def RotateBall():
    #draw a 3x3 ball on display & move it around
    ShowRoutine("Rotate Ball")
    SetPattern(0x0E,0x0E)
    RotateDisplay("right")
    RotateDisplay("left")
    RotateDisplay("up")
    RotateDisplay("down")

def MultiplexDisplay (z,count=40):
    # call this routine with Z, the image to display on the matrix
    # Z is a list containing 8 bytes (8x8=64 bits for 64 LEDs)
    # z[0] is data for the top row of the display; z[7] is bottom row.
    # count is number of times to cycle the display
    for count in range(0,count):
        for row in range(0,8):
            SetPattern(1<<row,z[row])       #quickly display each row

def TrySomeRoutines():
    Pulse1()
    Pulse4()
    SingleRowCylon()
    MultiRowCylon()
    AllRowCylon()
    SingleColumnCylon()
    MultiColumnCylon()
    AllColumnCylon()
    Marquis()
    RotateBall()
    Figure8()
    PawCircle()
    RandomPatterns()
    RandomPixels()

#
#   Main Program here.
#

print "Pi Matrix library test starting"
bus = smbus.SMBus(1)            #initialize the I2 bus; use '0' for
                                #older Pi boards, '1' for newer ones.
EnableLEDS()                    #initialize the Pi Matrix board
TrySomeRoutines()
DisableLEDS()                   #dont leave any LEDs on.
print "Done."
```