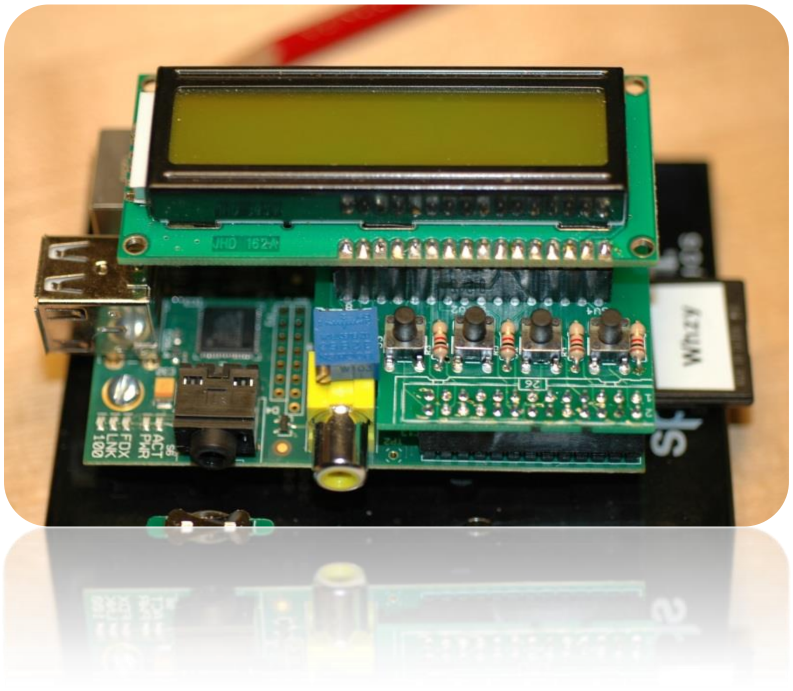


Beginner's Guide to the PI LCD

Part 3: Graphics Intro

Bruce E. Hall, W8BH

1) INTRODUCTION

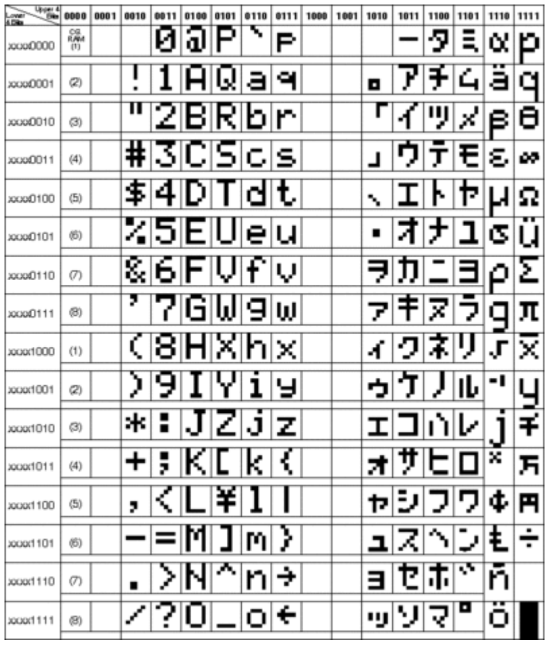
In Part 1 and Part 2 of this series, we learned how to check the switches and display text on the LCD board (available from mypishop.com). Now it's time to learn how to display more complex data, including graphics. You may have purchased either a 16x2 or a 20x4 display with your kit. In this write-up I'll be using the 20x4 display, but a 16x2 will work just as well.

2) THE HD44780 CHARACTER SETS

Here is a list of characters that our LCD can display. Numbers, uppercase letters, and lowercase letters are in their standard ASCII positions. There is a block of unused character positions in the middle at addresses 0x80 to 0x9F. On the right is a set of Japanese katakana characters. This 'A0' character set is the set available on my display.

Some controller chips are burned with a different character set, one that contains European and Cyrillic characters instead of the Japanese ones. Interesting! Which one do you have? Let's find out.

The easiest way is to send a single character to the screen, and see what it looks like. For example, character 0xE0 on my display is the Greek letter alpha (α). On a chip with the A2



Upper 4 Bits / Lower 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	p
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	β	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	g	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	⁻¹	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	千
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	ˣ	万
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	円
xxxx1101	(6)		-	=	M	]	m	}			ュ	ス	ヘ	ン	£	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	ñ	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	゜	ö	█

character set, 0xE0 is an a-grave (à).

```
#determine the character set
SendByte(0x01)        #clear the display
SendByte(0xE0,True)   #Either alpha (A0) or a-grave (A2)
```

Does your controller chip have A0 or A2? Now, just for fun, let's fill up the display with characters. I set up this routine for 4 rows of 15 characters each, giving me extra space for a text label.

```
def FillChar(code):
      for count in range(60):
            UpdateCursor(count)
            SendByte(code,True)

def UpdateCursor(count):
      if count==0:
            GotoLine(0)
      elif count=15:
            GotoLine(1)
      elif count=30:
            GotoLine(2)
      elif count=45:
            GotoLine(3)
```

Now try a call to FillChar(0xE0), and watch your display fill up with alphas. UpdateCursor makes sure that you go from line to line at the appropriate times. Rather inelegant looking, isn't it? How about something more compact?

```
def UpdateCursor(count):
      if (count%15)==0:
            GotoLine(count/15)
```

Is this any better? I stuck with the first version, because it is fast, straightforward and easy to understand. You choose!

Finally, let's mix it up a bit and display more of the character set. You can sequence though all 255 character codes, but there are two large empty blocks in A0 that don't have characters. A function for getting the next available character will get rid of those empty blocks:

```
def GetNextCharacter(code):
      if (code<0x20) or (code>=0xFF):        #remove first empty block
            code = 0x20
      elif (code>=0x7F) and (code<0xA0):     #remove second empty block
            code = 0xA0
      else:
            code += 1                        #OK to get next character
      return code
```

Now create a routine that looks like FillChar, except for an added call to GetNextCharacter. To slow down the action, delay after each character for a fraction of a second:

```
def FillScreen(code,delay=0):
      for count in range(60):
            UpdateCursor(count)
```

```
                SendByte(code,True)
                code = GetNextCharacter(code)
                time.sleep(delay)

def CharTest(numCycles):
        for count in range(numCycles):
                rand   = random.randint(0,255)
                firstChar = GetNextCharacter(rand)
                FillScreen(firstChar,0.03)

CharTest(5)    #Five screens of characters
```

3) TIMING IS EVERYTHING

If you are like me, you'll watch some slow character screens for a while, then remove all delays to see how zippy it is. You ought to be able to send 60 characters to the display without any noticable lag, right? In fact, you ought to be able to send 600 characters without any lag. I was a bit surprised to discover otherwise. Try the following code:

```
def NumberTest(delay=1):
#show an almost-full screen (60 chars) of each digit 0-9
        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 = 0.50
        for count in range(numCycles):
                startTime = time.time()
                NumberTest(pause)
                elapsedTime = time.time()-startTime
                elapsedTime -= pause*10
                print "  elapsed time (sec): %.3f" % elapsedTime

TimeTest()
```

You will need to import the time module for this one. NumberTest will send sixty '0' characters to the screen, then sixty '1' characters, all the way to '9'. It will briefly pause between each set to that you can verify that the numbers displayed correctly. TimeTest runs this sequence three times, and measures the time for each test to run.

The key to measuring time is the time.time() function. We call it twice: once before and once after the call to NumberTest. The difference between the two values is the amount of time that NumberTest took to execute. I run it three times to see the variability: our python code shares CPU time with other processes, and may take more or less time depending on what else is running.

The elapsed times will show on your screen. How many seconds does it take? On my system, a call to NumberTest averages about 4.50 seconds. That's pretty fast for a human, but slower than slow for our raspberry pi. Let's see what we can do to speed it up.

4) FASTER, FASTER

The functions in Part 2 included low-level routines for sending data to the LCD. One of those routines included timing delays required by the LCD. The routines worked, and I was happy. But now I wonder: can we shorten any of the delays? And if so, will it speed up our data transfer to the LCD? We now have a tool, TimeTest, that will measure time and give us a visual indicator of success.

Let's look at the routine with the most delays: PulseEnableLine. Recall that each byte of our data is sent to the controller as two 4-bit nibbles, and we clock each in a brief pulse on the enable (LCD_E) line:

```
def PulseEnableLine():
       mSec = 0.001
       time.sleep(mSec)                #DELAY (A): our first 1 mS wait
       GPIO.output(LCD_E, GPIO.HIGH)
       time.sleep(mSec)                #DELAY (B): our second 1 mS wait
       GPIO.output(LCD_E, GPIO.LOW)
       time.sleep(mSec)                #DELAY (C): our third 1 mS wait
```

This routine contains 3 mS of waiting time, and it is called twice per byte. In Part 2 I said we could halve those waiting times, reducing our waiting from 6 mS to 3 mS per byte. It works. But when I try to reduce those times to 0.1 mS (100 microseconds), my display falters.

When in doubt, RTFM (read the manual). On page 49, the HD44780 datasheet gives the timing for the LCD_E line as 450 nanoseconds. The controller needs less than half a microsecond of pulse! So why would our display falter at 100 microseconds? OK, RTFM again. Some commands require much more execution time than others. In particular, the Clear Screen and Return Home instructions each require a whopping 1.52 mS to execute – 40 times longer than the other commands. We call the Clear Screen command in our initialization sequence. Modify that command, adding in a delay:

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

Now we can go back to PulseEnableLine and reduce the delays. Here is a table of TimeTest results when I reduce the A, B, and C delays:

Condition	TimeTest Result
All delays = 1 mS	4.50 seconds
All delays = 0.5 mS	2.54
All delays = 0.1 mS	1.00
Remove A&C, B=0.1 mS	0.45
Remove A&C; B=0.001 mS	0.35
Remove A, B, & C	0.13

Wow! By removing the delays we have improved performance 35-fold. Why don't we need the delays? Because Python is slow, and each Python statement takes longer than the required pulse time. Try it and see.

5) CUSTOM CHARACTERS

The LCD controller is great for ready-made alphanumeric characters: send the code for a character in ROM, and it is displayed on the LCD. But there is also a very small amount of RAM reserved for custom characters. The controller can hold up to 8 characters of your own design. To create and display these characters, use the following procedure:

- Design your 5x8 character
- Code it into 8 bytes, one byte for each row
- Save the data in the controller's 'character generator RAM'
- Repeat above steps for up to 8 characters
- Display the character by sending the corresponding code (0x00 through 0x07)

Let's try a useful example: a battery status indicator. We will create seven symbols, going from no-charge to full-charge. Fill in the outline of the no-charge symbol, assigning a '1' to a filled square and '0' to a clear square.

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

0b01110 = 0x1E
0b11011 = 0x1B
0b10001 = 0x11
0b10001 = 0x11
0b10001 = 0x11
0b10001 = 0x11
0b10001 = 0x11
0b11111 = 0x1F

Here is the battery symbol on a 5x8 grid. There are 8 rows, and each row has 5 columns. The green box shows how we convert the binary ones and zeros of each row to their hexadecimal equivalent. Our 8-byte representation for the symbol, from top to bottom, is the list [0x1E, 0x1B, …, 0x1F]

To create a full-charge battery, just fill in the middle rows. As you fill each middle row, its value will change from 0x11 to 0x1F. A list of all of these battery symbols looks like this:

```
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)
]
```

Each row of this list corresponds to a battery symbol, from the no-charge symbol on top, to the full-charge symbol on the bottom.

6) LOADING THE CG-RAM

Now that we have our artwork encoded, it's time to load it into the character-generator RAM. Remember that we have 8 slots (of 8 bytes each), so our list of 7 symbols will fit. To load a single character, we send a 'LOADSYMBOL' (0x40) command which includes the
start address in its lower 6 bits. For example, if we want to store an 8 bit character for position 0x02, the command will be 0x40 + 0x10 = 0x50.

Character Position	**CG-RAM Address**
0x00	0x00 – 0x07
0x01	0x08 – 0x0F
0x02	0x10 – 0x17
0x03	0x18 – 0x1F
0x04	0x20 – 0x27
0x05	0x28 – 0x2F
0x06	0x30 – 0x37
0x07	0x38 – 0x3F

Notice that the starting address is just the symbol position * 8. After we send this command, all we need to do is to send the 8 bytes as character data:

```
def LoadCustomSymbol(posn,data):
       cmd = LOADSYMBOL + posn<<3
       for byte in data:
              SendByte(byte,True)

def LoadSymbolBlock(data):
       for i in range(len(data)):
              LoadCustomSymbol(i,data[i])
```

Now that all of the battery symbols are loaded, you can display each one by sending characters 0x00 through 0x07 to the LCD. If you keep the cursor in the same position, you can animate the character to look like your cell phone 'battery is charging' symbol:

```
def AnimateCharTest(numCycles=8,delay=0.05):
      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
```

That's all for part 3. In the next part we will create some useful graph functions and make a large-digit clock.

7) PYTHON SCRIPT for PI LCD, PART 3:

```
#!/usr/bin/python

########################################################################
#
#    LCD2:  Learning how to control an LCD module from Pi
#
#    Author:  Bruce E. Hall  <bhall66@gmail.com>
#    Date  :  10 Mar 2013
#
#    This code assumes 20x4 display, but will run on 16x2 display.
#    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]

#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
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

#custom character-generator symbols
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)
]


#########################################################################
#
#   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 displaying 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)


########################################################################
#
#   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])
```

```
########################################################################
#
#   Basic HD44780/LCD Test Routines
#
#

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))
        else:
            ShowMessage(st)

def AnimateCharTest(numCycles=8,delay=0.1):
    LabelTest('Animation')
    LoadSymbolBlock(battery)                #get all battery symbols
    GotoXY(1,3)                             #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 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=0.03):
    #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


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

print "Pi LCD3 program starting."
InitIO()
InitLCD()
TimeTest()
CharTest()
AnimateCharTest()
print "Done."


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