LCDs & Things

By Jon Williams

For Nuts & Volts Magazine, Column 2, September 2009

It must have been 1994 when I discovered how much I enjoy character LCDs. Like so many others, I got started thanks to Scott Edwards and his articles here in Nuts & Volts. As soon as I had one LCD working I was hooked and have used them with the BS1, BS2, and the SX – so why not the Propeller? To my surprise there wasn't any particularly good 4-bit LCD code in the Propeller Object Exchange so I pulled together my own best stuff and ported it. When we combine the LCD with a mini digital joystick we have a nice little user interface for projects that move beyond the lab.

Perhaps it's just me, but I find it somewhat humorous that many consider video the standard visual output for the Propeller multiprocessor. Sure, it's really cool, but in my mind it's not very practical for portable applications – and sometimes I want a project to move off my desk! This is where character-based LCDs are eminently practical: they come in a variety of sizes, are low-powered, and are really easy to interface. For my own projects I tend to use the 4-bit interface that is defined by the Hitachi HD44780 specification; the standard which other vendors seem to follow without question. With a 4-bit buss, three-control lines, and an output to control the [optional] LCD backlight, we can squeeze the whole works into eight pins (convenient for micros that have 8-bit ports).

Figure 1 shows the schematic for the 4-bit LCD connections. If it seems like you're having a case of déjà vu, don't worry, this schematic is identical to what we used in the SX28-based intervalometer project (*Nuts & Volts*, March 2009). The only difference is the addition of current limiters on the data pins. Why add these resistors? Well, the Propeller is a 3.3v device and the LCD is a 5v device. When the Propeller is reading the LCD buss the protection diodes on the Propeller IO pins will clamp the 5v down to a safe level; these resistors minimize the current through those diodes to protect them.

Before I move on, let me point out something about the LCD called out in the BOM (Hantronix HDM08216-3-L30S): when I plugged it into the board the backlight turned on. This was odd as there is a transistor circuit to control the backlight and it should not go on without an explicit command. Well, after I proved the control pin on the Propeller was fine and that the transistor circuit was okay, I checked continuity between the cathode (K) pin and ground – they were shorted together on the LCD. I was livid – the spec sheet for the LCD does not indicate that the backlight cathode is connected to ground!

So, despite what we've all been told, I went to bed mad and the next day added a second transistor to the circuit to create a high-side driver that could switch the anode (A) pin with a high-output from the Propeller. Then I noticed something on the LCD: a small solder jumper between the cathode connection and ground. So I heated up the soldering iron, removed it, and BAM!, everything is working fine with the original circuit. I probably should have noticed that the night before but I was tired – lesson learned.

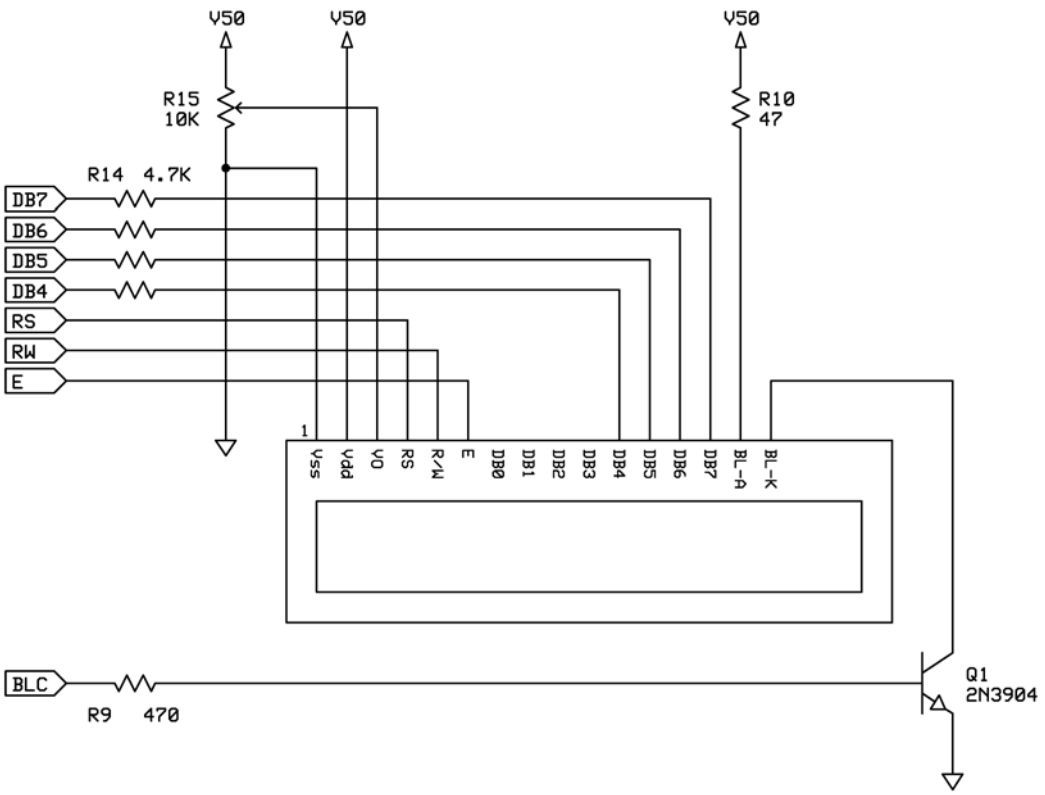

Figure 1: LCD Connections

So… if you use the backlit LCD that I call out in the BOM you'll want to make sure that the little jumper (marked J1 on the LCD I have) is removed so that you can control the cathode pin with the single-transistor circuit shown in Figure 1.

Creating an object for the LCD is really quite easy, and as there are no critical timing or "background" requirements we can do it all in Spin. Let's begin with the initialization of the LCD. We're going to call this method with the first pin of an eight-pin group and pass the number of columns and rows for the LCD.

```
pub init(blpin, cols, lines) | okay

  if (blpin > 20)
    okay := inuse := false

  else
    finalize
    bl  := blpin
    e   := blpin + 1
    rw  := blpin + 2
    rs  := blpin + 3
    db4 := blpin + 4
    db7 := blpin + 7

    if lookdown(cols:8,16,20,24,32,40)
      lcdx := cols
    else
      lcdx := 16

    if lookdown(lines : 1, 2, 4)
      lcdy := lines
```

```
    else
      lcdy := 2

    outa[db7..bl] := %0000_0000
    dira[db7..bl] := %1111_1110
    lcdinit
    okay := inuse := true

  return okay
```

The code starts by checking the backlight control pin to ensure that the group will not collide with the Propeller I2C and programming pins (28 – 31). If the backlight pin number is okay then the rest of the pins are defined. Some will notice that I maintained the order used by the BASIC Stamp LCD functions – some habits die hard!

With the LCD pins defined the columns and lines parameters are validated. As I stated earlier, character LCDs come in variety of sizes and yet there are standards. As 16x2 LCDs seem to be the most common these values are used as the default settings if a bad parameter is passed. The *lcdx* and *lcdy* variables will be used by other methods to make sure we don't attempt to write to an area of the LCD that isn't present. With the parameters all set the LCD pins are cleared and all but the backlight control pin are set to outputs; the backlight pin is left as an input for the time-being (you'll see why in just a moment).

What we've just worked through is the top-level initialization; after the buss pins have been setup then we call the low-level initialization that puts the LCD into 4-bit mode and makes it ready to use.

```
pri lcdinit

  waitcnt((20 * MS_001) + cnt)
  outa[db7..bl] := %0011_0000
  blipe
  waitcnt((5 * MS_001) + cnt)
  blipe
  waitcnt((150 * US_001) + cnt)
  blipe
  outa[db7..bl] := %0010_0000
  blipe
  if (lcdy > 1)
    cmd(%0010_1000)
  cmd(%0000_0110)
  dispCtrl := %0000_1100
  cmd(dispCtrl)
  cmd(CLS)
```

If you've ever used an LCD with a BASIC Stamp or the SX then this code should look really familiar; in fact, I copied my SX/B source code for this method into the Propeller editor and then made the [minor] adjustments to Spin. Note that this method is declared as private (**pri**); what this means is that the method cannot be "seen" outside the LCD file, even by an object declared as an LCD. So, private methods are for use only within an object file; public (**pub**) methods may be used internally and externally.

The initlcd() method follows the conventions for 4-bit LCD initialization that are detailed in the Hitachi HD44780 data sheet. As you can see we use *lcdy* to control multi-line initialization. And per usual practices the LCD is set to auto-increment after a write operation. Finally, we're initializing and using a variable called *dispCtrl*; this will keep track of the display (on or off) and cursor setting (none, underline, blink, or both) so that we can change either without upsetting the others. The starting value of *dispCtrl* turns the display on and the cursor off.

Once the LCD is initialized to 4-bit mode we don't have to write directly to the buss, we can use the cmd() method; this is one of two ways to send information to the LCD. This method sends a command (e.g.,

clear the LCD, move the cursor, etc.). Another method that we'll use frequently is called out(); we'll use this to write characters. I deliberately named these methods to make the translation of older PBASIC programs as simple as possible.

There is a small, yet critical difference between the cmd() and out() methods: the status of the RS (register select) line to the LCD. For a command the RS line must be set to 0, for a character the RS line must be set to 1.

```
pub cmd(c)

  waitbusy
  outa[rs] := 0
  wrlcd(c)


pub out(c)

  waitbusy
  outa[rs] := 1
  wrlcd(c)
```

One of the aspects of LCD interfacing that many programmers give up on is reading the busy flag. We really should read this flag as this enables us to write a new command or character as soon as the LCD is ready; inserting an arbitrary delay just slows everything down and reduces throughput. With a 4-bit buss it takes a little work, but as you'll see it's really not too difficult.

```
pub waitbusy | addr

  dira[db7..db4] := %0000
  outa[rs] := 0
  outa[rw] := 1

  repeat
    outa[e] := 1
    waitcnt((5 * US_001) + cnt)
    addr := ina[db7..db4] << 4
    outa[e] := 0
    waitcnt((5 * US_001) + cnt)
    outa[e] := 1
    waitcnt((5 * US_001) + cnt)
    addr |= ina[db7..db4]
    outa[e] := 0
  while (addr & %1000_0000)

  return (addr & $7F)
```

The method starts by making the data pins inputs and placing the LCD into command (RS = 0) and read (RW = 1) mode. We can read one nibble of the cursor address at a time by "blipping" the enable (e) pin. The high nibble is read first so you see that we have to shift it left by four bits. Then the lower nibble can be OR'd onto the *addr* variable; if bit7 of *addr* is set then the LCD is busy with the last command. The address scan is embedded a **repeat-while** loop that will run until the busy flag clears. Should we ever need to know the address of the cursor that information is returned by this method.

Before I forget, let me point something out regarding the use of **waitcnt** to do sub-millisecond timing: Spin is an interpreted language and as fast as it is, the Propeller cannot accept a delay value of less than five when using the syntax described here (instruction overhead makes the actual delay longer than 5us). I point this out so that you're not tempted to shorten the delays after reading an LCD spec sheet. If we get a **waitcnt** rollover the delay will go from what we wanted (a few microseconds) to almost a full minute – yes, it takes that long to run through the 32-bit system counter at 80 MHz.

Back to writing to the LCD… when we've determined the LCD is not busy we can set the RS line as required (low for a command, high for a character) and then call the wrlcd() method.

```
pri wrlcd(b)

  dira[db7..db4] := %1111
  outa[rw] := 0

  outa[db7..db4] := (b & $F0) >> 4
  blipe
  outa[db7..db4] := b & $0F
  blipe
```

As we're writing to the LCD the data pins are set as outputs and the LCD to write mode (RW = 0). Again we have to transfer four bits at a time, starting with the high nibble. After a nibble has been placed on the LCD data buss it is transferred by blipping the Enable (e) pin.

When you download the files you'll see that there are a lot of useful methods in the LCD object and as most are self-explanatory there is no reason to go into detail here. One method I do want to discuss, though, is called scrollstr(); we can use this to scroll a string through "window" in the LCD.

```
pub scrollstr(x, y, w, ms, pntr) {
} | okay, p, len

  okay := false

  if (x => 1) & (x =< (lcdx + 1 - w))
    if (y => 1) & (y =< lcdy)
      len := strsize(pntr)
      if (len => w)
        repeat (len - w + 1)
          p := pntr
          moveto(x, y)
          repeat w
            out(byte[p++])
          waitcnt((ms * MS_001) + cnt)
          pntr++
       okay := true

  return okay
```

We need to pass the column (x) and line (y) that defines the left edge of the scroll window, the width (w), the delay (in milliseconds) between moves, and a pointer to the string. As you can see the location and width values are validated before we attempt the scroll; we have to make sure that the defined window will fit onto the LCD (based on its width) and that we've selected a legal line. We also need to ensure that the string is at least as long as the window.

If everything checks out then a couple nested **repeat** loops do the work. The outer loop controls how many scroll events are required; this is based on the width of the scroll window and the length of the string. Inside that loop a pointer is set to the first character to print, the cursor is moved to the left edge of the scroll window, then "w" characters are printed to fill the window. After the delay the character pointer is advanced and the inner loop is run again.

If we want to have a string scroll on cleanly (i.e., start with an empty window) then we need to prefix the string with spaces, at least as many as the width of the window. Conversely, if we want the string to scroll off cleanly then we need to append a number of spaces to the string to take care of that. It won't take more than a few minutes of play with this method to understand how to take full advantage of it – and

remember that pressing F10 in the Propeller IDE downloads very quickly to RAM. And if right-to-left is not enough, there is a method called rscrollstr() that does the same thing in reverse, i.e., left-to-right.

Green Backlight Control

Nope, I don't mean green as in the color, I mean green as in energy conservation. The LCD object allows for setting or clearing the backlight, but these are static controls; the backlight is either on or it's off. What if we create a device that uses a backlit LCD that we're not going to look at all the time – wouldn't it be a good idea to kill the backlight when we don't need it to be operating? You bet, and we can do just that with a very small Assembly program.

Here's the whole works:

```
dat

                org     0

oneshot         test    osidle, osidle      wc
                muxc    outa, osmask
                or      dira, osmask

                mov     ms1timer, MS_001
                add     ms1timer, cnt

osloop          rdlong  ostimer, ospntr     wz
        if_z    muxc    outa, osmask
        if_nz   muxnc   outa, osmask
        if_nz   sub     ostimer, #1
                wrlong  ostimer, ospntr
                waitcnt ms1timer, MS_001
                jmp     #osloop


MS_001          long    0-0

osmask          long    0-0
osidle          long    0-0
ospntr          long    0-0

ostimer         res     1
ms1timer        res     1

                fit     492
```

Before I explain the code let me tell you about the variable section. The first thing you probably noticed are symbols declared as **long** that have a strange value: 0-0. I don't know who started this convention but it is generally accepted among Propeller programmers that this defines a value that will be set or modified by another instruction.

This is possible because all programs are running in RAM; any piece of data – even instructions – can be modified on the fly. This is tremendously powerful and tremendously dangerous at the same time if not handled carefully. In our case we're going to modify these symbols from a Spin method. Until the PASM program is launched with **cognew** it is just data to the Spin program and can be changed at will. Making the modifications in Spin is a little easier than doing the setup in Assembly, especially when we want to use fractional values (where division is required) of the system clock frequency.

Here's the Spin code that sets up the one-shot object parameters and launches it:

```
pub init(p, idle, ms) | okay
```

```
finalize

if (p => 0) and (p =< 27)
  MS_001 := clkfreq / 1_000
  osmask := 1 << p
  osidle := (idle > 0) & 1
  ospntr := @osDuration

  okay := cog := cognew(@oneshot, 0)+1
  run(ms)

else
  okay := false

return okay
```

We have to pass the pin number, the idle state (0 or 1) and the number of milliseconds for the initial pulse. If the pin number is legal then we modify parameters that will be used by the PASM program. The first is MS_001 that is the number of system clock ticks in one millisecond. A bit mask is created for the pin, the parameter called *osidle* is set to 0 or 1 and, finally, the pointer to the one-shot timing variable (which is in the hub) is initialized.

Okay, let's go look at the PASM section. On entry we test the *osidle* value which was previously forced to zero (idle state is low) or one (idle state is high). The reason we've forced a non-zero value to one is that we're going to use the carry flag to save the idle state. This is accomplished by using **test** on *osidle* against one. The **test** operator works just like **and** but it does not affect the destination. When **wc** is used with **test** the carry flag will be set (1) if there is an odd number of bits in the **test** result. At this point the carry flag will match the idle state of the one-shot pin: 0 for low, 1 for high.

Next we use **muxc** to write the value of the carry flag to the output pin which is defined in *osmask*. With the idle state set the pin is made an output. Since we won't use **wc** in any other instructions the carry flag will be maintained through the run of the program and we don't have to retest the idle state. Once you get used to the idea of having control over the carry and zero flags Propeller Assembly can be quite fun.

The next step is to initialize a one-millisecond timer and then drop into the main loop. The first task is to read the one-shot timing from the hub. If the time is zero (**if_z**) we return the pin to its idle state, otherwise we set it to the active state. If the time is greater than zero (**if_nz**) then it gets decremented, we let the one-millisecond timer expire and then we write the timing value back to the hub.

Why bother writing the timing value back to the hub? Well, if we keep the timing variable in one place then both sides can modify it at will; this allows use to truncate a one-shot command if we want to.

So, how would we use the one-shot control with the LCD? Easy: if a button press is detected (see below) then we'll (re)set the one-shot timer. As long as we're busy with the buttons the backlight will stay lit; with no input after some pre-determined period (I use five seconds) the backlight will go off and we've reduced current consumption by about 140 milliamps. This is especially useful in battery-powered applications

The Joy of a Joystick

In the December 2007 issue of *Nuts & Volts* Joe DeMeyer described an intervalometer that used a mini digital joystick; the stick has four direction switches plus a center switch. I bought one of these little dudes and it's really cool so I incorporated it into the LCD UI. With the joystick and one additional pushbutton we have a flexible way of navigating menus and updating information displayed in the LCD. Figure 2 shows the schematic for the joystick/button interface. All the inputs are pulled low and when active read as "1" on the corresponding Propeller pin.

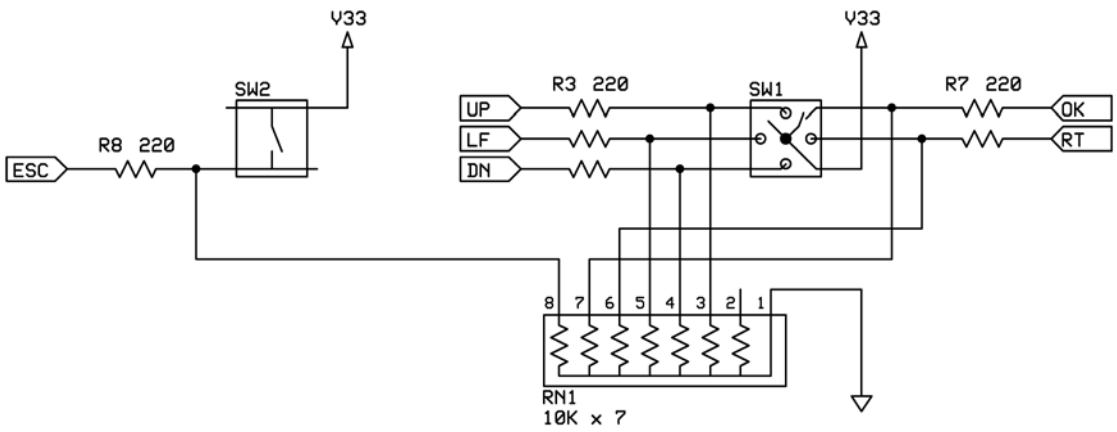

Figure 2: Joystick and Button Connections

We all recognize that debouncing switch inputs is a good idea and yet this can take precious time from a program if done in the foreground. So, let's create a debouncing program and run it in a separate cog, shall we?

For flexibility we're going to pass a mask that defines which pins to scan; a "1" bit in the mask means we'll scan that pin, a "0" bit means the pin is ignored. We'll also pass the debounce timing duration (I tend to use 25ms), and the logic of the inputs; "1" for active-high (as on the LCD UI board), "0" for active-low (as on the PPDB).

Here's the PASM code that handles the debouncing.

```
dat

                org     0

debounce        andn    dira, scanmask

                mov     dbtimer, SCAN_DELAY
                add     dbtimer, cnt

newscan         mov     scanresult, scanmask
                mov     scancount, scantime
                shl     scancount, #3

                test    scanmode, #1        wz
scan    if_nz   and     scanresult, ina
        if_z    andn    scanresult, ina
                waitcnt dbtimer, SCAN_DELAY
                djnz    scancount, #scan

                wrlong  scanresult, scanbits
                jmp     #newscan
```

On entry we ensure the selected pins are inputs, setup a timer and then drop into the scan. The scan works by initializing the result to the all pins active – easily done by copying the mask into the result. Then we test the mode and if active high the inputs (**ina**) are ANDed with the result; any pin still high will remain a "1" bit in the result – if or until it goes low during the scan. If a pin does go inactive at any point in the scan the use of AND will cause that bit to stay inactive (0) in the result.

If the scan mode is zero then we AND the inverted state of the pins with the result; we're able to do this with the (very convenient) **andn** operator. With the inputs scan out of the way the timer is allowed to expire and the scan count is decremented. When the scan window is completed the result is written back to the hub and we start a new scan. How easy is that?

Of course you've noticed that the result bits are active-high ("1" means the input is active), even if the physical pins are wired as active-low. I think this makes the higher level code easier to deal with, especially when creating masks for individual pins.

There are two methods in the high-level interface of the debounce object; one that returns true if *any* of the pins in the mask are active and another that lets us get the state of one or more pins from the debounced result; both methods will be used in the demo program.

Bi-Color LED Redux

Since the button inputs only require six bits it made sense to pop a bicolor LED onto the board – this neatly fills out a group of bits and can provide important information that one might miss on the LCD unless right on top of it. I've had a lot of time to work with the bi-color LED and have made some adjustments.

One of the changes is that it now supports 2- and 3-lead LEDs. The latter usually has a common cathode for the internal LED chips so completely extinguishing the LED means that both pins must be turned off. The bigger change is that the PWM aspect is separated for each color chip. This allows us to balance red and green brightness levels for either style LED, and find the best balance for yellow without having to dig into the PASM code. Have a look at the new object; I think you'll like it.

Figure 3 shows the connections for a 3-lead, bi-color LED. If you decide to go with a 2-led type then you can change one of the resistors to a jumper.

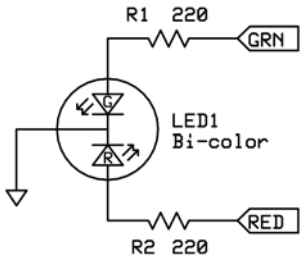

Figure 3: Bicolor LED Connections

Putting It All Together

Okay, we have an LCD object, a one-shot object that's useful for controlling the LCD backlight, a debounce object for scanning the joystick and button inputs and, finally, an update of the bi-color LED. The LCD UI is an integrated module with pre-defined pin connections so it makes sense to wrap these discrete objects into one master object that we can use in future projects.

Although we use the term "object" Spin is not an OOP language like Java or Python, that is, when we create a new object from others, the methods from the source objects are not inherited by the new object. That's the bad. The good is that we can determine which of the source object methods we want to expose, can rename them if we like and, of course, we're free to create new methods from those that exist in any of the source objects.

For example, you'll find a cls() method in the *lcd_ui* object.

```
pub cls

  lcd.cmd(lcd#CLS)
```

As you can see this is actually a call to the LCD's cmd() method using the LCD's CLS constant. We'll put this to work in the demo program. As always, I encourage you to experiment with the demo program and then really put it to use. First up for me is connecting an IR LED to the LCD UI expansion buss and porting the intervalometer program that I originally wrote for the SX28.

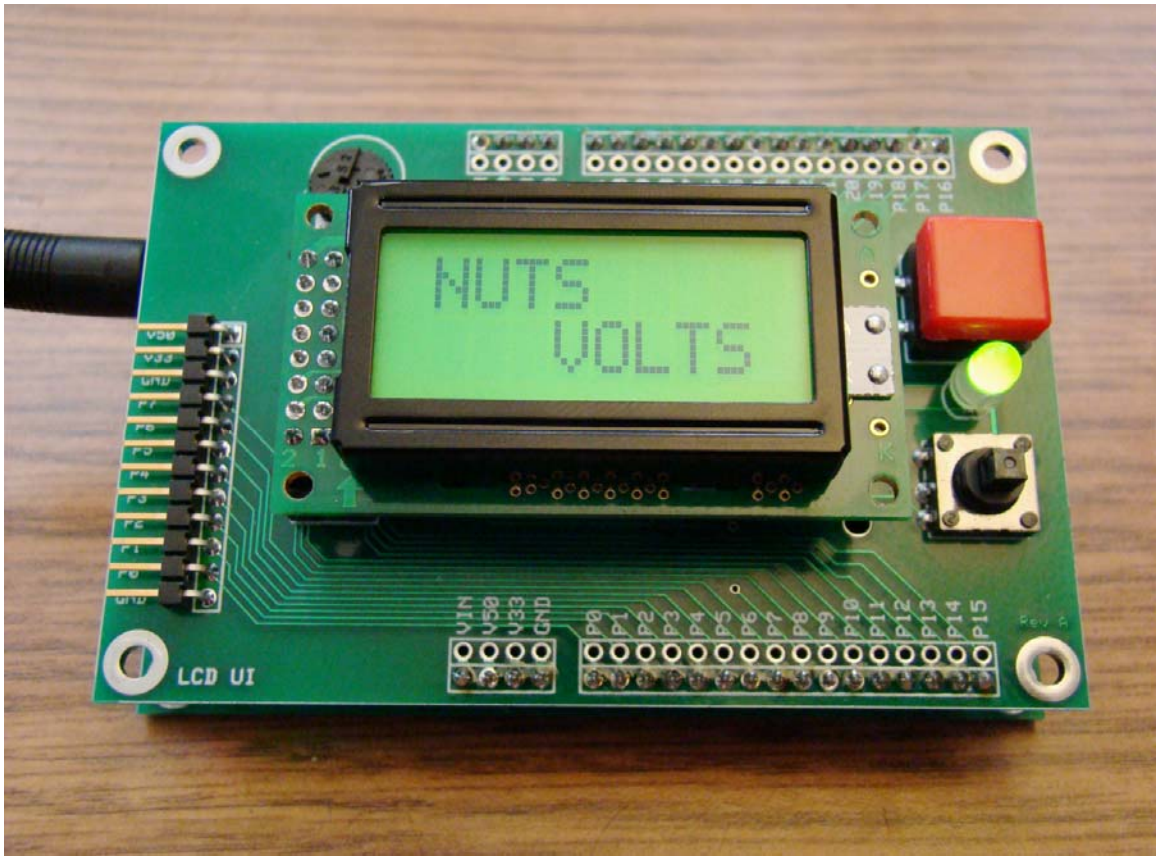

Figure 4: Completed LCD UI

I'm a Gadget Gangster!

If there is a cooler name for a technology-centric web site on the Internet, I haven't seen it – *www.gadgetganster.com* has a great name and really fun stuff. Gadget Gangster is run by a nice guy named Nick McClanahan who has taken me up on my "Go forth and be prosperous!" call vis-à-vis projects I create for my columns.

Nick really liked the Propeller Platform and after talking with Ken Gracey at the Propeller Expo in July, it was decided that Gadget Gangster was a better home for the Propeller Platform kit and future kits designed for it (like the LCD UI board). Now, this is not to say that Nick is going to make a kit for every project that appears in this column, but where there's broad interest in a given circuit he probably will.

100 MHz Propeller

Until recently the top speed of the Propeller chip has been 80 MHz. This is derived by using a standard 5 MHz crystal and a PLL setting of 16x. Well, if 80 MHz is good then 100 MHz is better, right? Absolutely! An enterprising Propeller programmer named Bill Henning has done us all a favor by having custom 6.25 MHz crystals made – this crystal and the 16x PLL setting gives us 100 MHz, a 25% boost in Propeller speed!

One advanced user who has pushed the Propeller even faster suggests changing the Vcc bypass cap near the crystal to a 4.7 to 10uF tantalum; this is an easy change and the layout of the Propeller Platform accommodates this with no trouble. You can buy the crystals directly from Bill's web site (*www.mikronauts.com*), from Parallax, and from Gadget Gangster.

I think that's about enough, don't you? Have fun with the LCD UI and until next time, here's to *spinning and winning* with the Propeller.

Bill of Materials

LCD	8x2, backlit	Mouser HDM08216L-3-L30S
LED1	Bi-color	Mouser 78-TLUV5300
Q1	2N3904	Mouser 610-2N3904
R1	220	Mouser 291-220-RC
R2	220	Mouser 291-220-RC
R3	220	Mouser 291-220-RC
R4	220	Mouser 291-220-RC
R5	220	Mouser 291-220-RC
R6	220	Mouser 291-220-RC
R7	220	Mouser 291-220-RC
R8	220	Mouser 291-220-RC
R9	470	Mouser 291-470-RC
R10	47	Mouser 291-47-RC
R11	4.7K	Mouser 291-4.7K-RC
R12	4.7K	Mouser 291-4.7K-RC
R13	4.7K	Mouser 291-4.7K-RC
R14	4.7K	Mouser 291-4.7K-RC
R15	10K	Mouser 652-3352T-1-103LF
RN1	10K x 7	Mouser 71-CSC08A01-10K
SW1	4x+1 NO	Mouser 688-SKQUCA
SW2	NO	Mouser 101-TS6211T3202AC-EV
SW2-CAP	black, 12x12mm	Mouser 101-0110-EV
X1	0.1 M-STRT	Mouser 517-6111TG
X2	0.1 M-STRT	Mouser 517-6111TG
X3	0.1 M-STRT	Mouser 517-6111TG
X4	0.1 M-STRT	Mouser 517-6111TG
X5	0.1 Box Header, 16p	Mouser 517-30316-6002
X6	0.1 M-R/A	Mouser 517-5111TG
XLCD	0.1 strip socket	Mouser 517-974-01-16
PCB		ExpressPCB