Column #102 October 2003 by Jon Williams:

This is Your Brain on Stamps

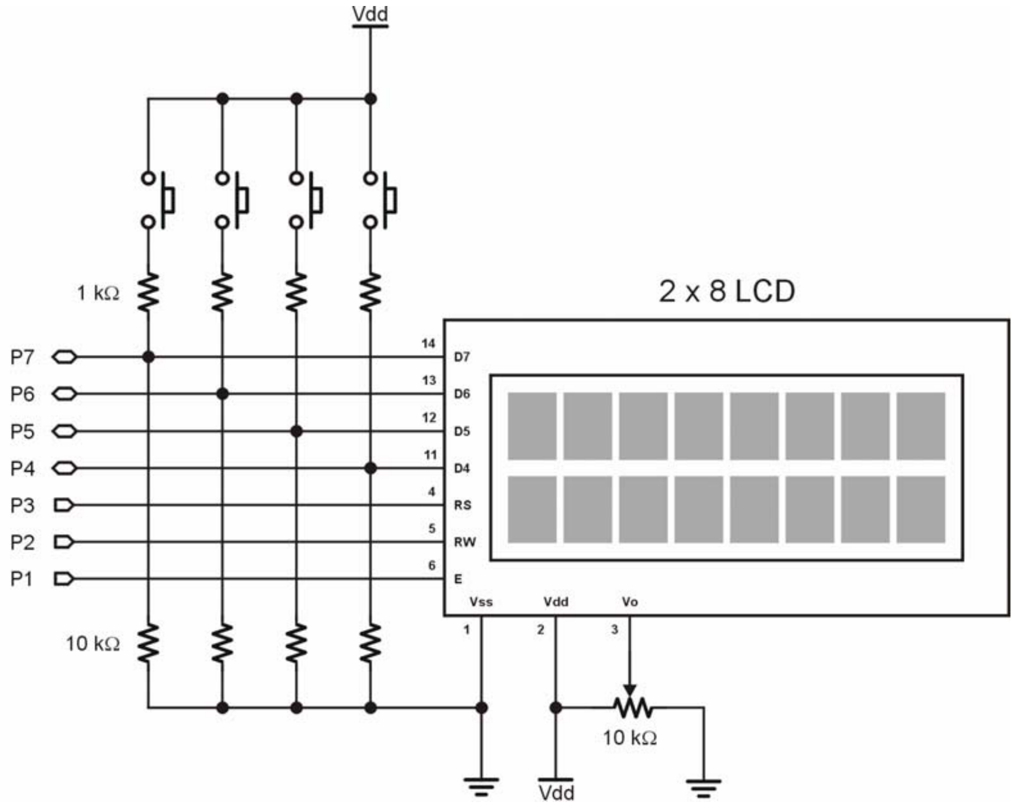
Since you read this magazine, and you're reading my column, you're probably a lot like me: smart, good-looking, a blast to hang-out with ... okay, okay, I was kidding – except for the smart part. That you read Nuts & Volts makes you very smart. What I should have said is that, like me, you're probably a gadget freak. Am I right?

The scariest element of this single-guy's home is not the rabid dust-bunnies living under my bed, it's the growing collection of "Wow, that looks neat..." gadgets that are now occupying that space. I guess if I just went to sleep at a decent hour and stopped watching late-night infomercials I'd have a smaller collection of junk. Oh well....

While cruising the web a couple weeks ago I came across an ad for something I'd seen before and considered buying: a brain-wave synchronizer. This is, essentially, a set of goggles that hold lights (LEDs) that can be flashed at controlled rates and patterns. My first acting coach taught me relaxation and self-hypnosis techniques and I thought it would be fun to add a little electronic assistance. But after thinking about it for a few minutes, I came to the conclusion that I could build my own with a BASIC Stamp. And that's what I did.

Now, even if you're not interested in brain-wave synchronizing, this project may still interest you in that it's a pretty good example of a simple menu system. And, as long-time readers know, one of my favorite topics with Stamps is lighting controllers.

Figure 102.1: Parallax LCD AppMod Circuit



This project is, in effect, a mini light show for the eyes and brain. I could just as easily hook it up to a set of opto-relays and control holiday lights. Okay, let's jump into this dude.

Menus Made Small

Since I wanted to test my project for a month or so before committing to a permanent enclosure, I built it on a Parallax Board of Education (BOE) – and actually, there is nothing connected to the BOE breadboard. What I did use is the new LCD AppMod from Parallax. It plugs into the BOE AppMod connector and gives me a nice little two-line by eight-character display and four button inputs. Now, the LCD AppMod is very straightforward (see Figure

102.1) and you can certainly assemble your own with a standard LCD, a few resistors, and some normally-open push-buttons.

The clever design of this LCD/buttons circuit was done by our old pal [and very smart guy] Scott Edwards. What's great about it is that a button press will not interfere with data being sent to the LCD. With a button open, the signal from the Stamp to LCD will be felt across the 10K resistor on that line. So, what happens when a button is pressed? Well, if the signal from the Stamp is high (5v), there is no conflict since both the Stamp and the pushbutton are sourcing 5v to the bus line. When the output from the Stamp pin is low and a button is pressed, there will be a small amount of current (about 5 mA) through the 1K resistor, but the LCD will still see a low since it is connected directly to the Stamp.

And when we use the bus pins as inputs to read the switches? With the button open, the 10K pulls the pin to Vss (ground) so we get a low (0). When the button is pressed, the 1K and 10K form a voltage divider and the Stamp will see about 4.5 volts on the input – this is well above the high threshold for a Stamp pin.

A Program For All BASIC Stamps

Not long after the security gate controller project I got a nice e-mail from a reader who was having problems with the code running on a BS2sx (my code was tested on a BS2) because the BS2sx is faster than the BS2. Then, there was the issue with the BS2p family since they have built-in LCD commands.

While writing the LCD AppMod documentation I decided to take full advantage of the conditional compilation feature of the new PBASIC compiler. I've covered it a bit before, but not as extensively as I will here. By using this technique, we can be sure that the program will work for anyone – no matter what their favorite flavor of BASIC Stamp may be. Since we're using the LCD AppMod for my project I'm able to take advantage of that code, and add a few tweaks to give it a bit more flexibility.

Let's start from the top. While most programs set pins to be inputs our outputs, this one requires the LCD bus and button input pins to change on-the-fly, so we're going to create a set of definitions that make the program easy to read and maintain.

```
E               PIN     1
RW              PIN     2
RS              PIN     3
BusDirs         VAR     DIRB
BusOuts         VAR     OUTB
```

```
BusIns          VAR     INB
LEDs            VAR     OUTH
```

The first three definitions are LCD control pins and as you can see, we've used the `PIN` type definition where we used to use `CON`. Remember that `PIN` helps us when we have IO pins that can be inputs or outputs, depending on what's happening with the program. Using `PIN` prevents duplicate definitions for the input and output register bits.

But `PIN` only works on single pins, not on groups. So to keep things as clean as possible, we've aligned the LCD bus with one of the Stamp's standard IO groups (P4 – P7). By using aliasing, we're able to rename the various elements of IO group B for use in the program. Finally, we do the same thing with the `OUTH` group (P8 – P15) for the LEDs.

Figure 102.2: LED Circuit Mounted Inside Goggles

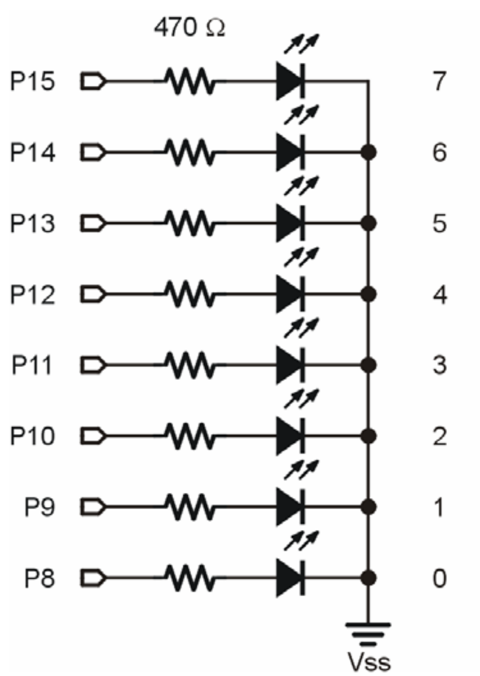


Figure 102.3: LED Physical Arrangement

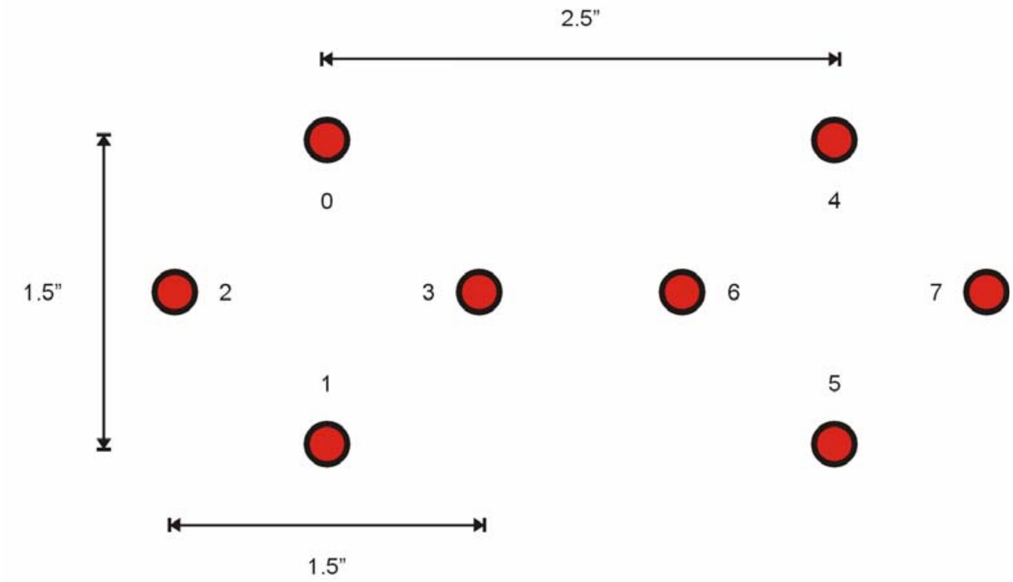


Figure 102.2 shows our LED circuit that is mounted inside the goggles, and Figure 102.3 shows how the LEDs are physically positioned.

Please allow me to stress that taking the time to do this kind of programming will save you lots of trouble later. Give your pins and pin groups names that are meaningful; not just for you, but for others who may come in contact with your program. I promise that this will save you more time than you might think you're wasting by being "fancy." Trust me, I know from experience.

Conditional Compiling

As I've told you before, the newest PBASIC compiler can – if we direct it – conditionally compile portions of code to be downloaded to our Stamp. The trigger for the conditional compilation can be automated or manual; it just depends on what we're trying to accomplish. For this program, we'd like the compiler to make things work no matter what BASIC Stamp we have plugged in. Let's see how.

In the Constants section we have this line:

```
#DEFINE _LcdReady = ($STAMP = BS2P) OR ($STAMP = BS2PE)
```

By using #DEFINE, we've created a compiler symbol. Compiler symbols are generally defined as True of False. If a symbol is not defined, but appears in program code, the compiler will assume it is False. In our case _LcdReady will be True if we're using a BS2p or BS2pe, otherwise it will be False. And keep in mind that the compiler actually polls the connected Stamp for its type before compiling and downloading the program. This keeps us from having a problem if our $STAMP definition is not matched to what we're actually connected to. The compiler will alert us and fix the $STAMP definition.

As a matter of style, I like to preface compiler symbols with an underscore character so that I know it's a compiler symbol and not a program constant. This keeps me from attempting to use them in normal program logic where an error would be generated. Where we can use them is in conditional compilation structures: #IF-#THEN and #SELECT-#CASE.

Okay, let's put the symbol to use. In the Initialization section of the program we go through the steps to get the LCD up and running with a four-bit bus and using two lines of characters. Here's the code:

```
LCD_Init:
  PAUSE 500
  #IF (_LcdReady) #THEN
    LCDCMD E, %00110000 : PAUSE 5
    LCDCMD E, %00110000 : PAUSE 0
    LCDCMD E, %00110000 : PAUSE 0
    LCDCMD E, %00100000 : PAUSE 0
    LCDCMD E, %00101000 : PAUSE 0
    LCDCMD E, %00001100 : PAUSE 0
    LCDCMD E, %00000110
  #ELSE
    BusOuts = %0011
    PULSOUT E, 3 : PAUSE 5
    PULSOUT E, 3 : PAUSE 0
    PULSOUT E, 3 : PAUSE 0
    BusOuts = %0010
    PULSOUT E, 3
    char = %00101000
    GOSUB LCD_Command
    char = %00001100
    GOSUB LCD_Command
    char = %00000110
    GOSUB LCD_Command
  #ENDIF
```

What's important to understand here is that only a portion of this code will be compiled and downloaded to the Stamp – what portion depends on the installed Stamp. If we're using a BS2p or BS2pe the first section (#THEN) will be compiled and downloaded allowing us to take advantage of the built-in LCD commands. If we're using any other BASIC Stamp, then the #ELSE section will be compiled and downloaded.

You can see this in action by using the Memory Map feature of the compiler and comparing the EEPROM use with one Stamp model versus another. Not surprisingly, the manual LCD code required by the BS2, BS2e, and BS2sx requires a bit more space than the built-in commands used by the BS2p and BS2pe.

There are two additional sections of code that use condition compilation.

```
LCD_Command:
  #IF (_LcdReady) #THEN
    LCDCMD E, char
    RETURN
  #ELSE
    LOW RS
    GOTO LCD_Write_Char
  #ENDIF
```

This first subroutine sends a command to the LCD. We use this to do things like clearing the LCD, moving the cursor, etcetera. Notice that there is a RETURN when we're using a BS2p or BS2pe, but when using other Stamps, we set the RS line low then jump to the LCD_Write_Char subroutine. The reason is that the mechanics of transferring a byte to the LCD are the same. The byte gets interpreted as a command or character based on the condition of the RS line. When using a BS2, for example, the RETURN from this subroutine call will actually be at the end of LCD_Write_Char. This is why we use GOTO in LCD_Command instead of another GOSUB – it keeps the RETURN stack cleaner and the program will actually run a bit more efficiently.

```
LCD_Write_Char:
  #IF (_LcdReady) #THEN
    LCDOUT E, 0, [char]
  #ELSE
    BusOuts = char.HIGHNIB
    PULSOUT E, 3
    BusOuts= char.LOWNIB
    PULSOUT E, 3
    HIGH RS
  #ENDIF
  RETURN
```

As you can see, LCD_Write_Char really does all the work. To keep things simple, we use a zero in the command field of the BS2p/BS2pe command. This means we won't do anything except write the character, which is what we do in the BS2/BS2e/BS2sx code. Also note that we set the RS line high when leaving the routine. This will cause the next call to this routine to default to character mode. The only time the RS line is taken low is when we pass through the LCD_Command subroutine.

Hypno-Goggles

Now that we can write to an LCD with any BASIC Stamp, let's get to the heart of our program and how we can use very small LCD to create an effective UI for our blinking LED goggles. Since the LCD is so narrow, one of the routines we'll find handy is a means of scrolling a long string through the window. Let's take a look:

```
LCD_Scroll_String:
  DO
    char = crsrPos
    GOSUB LCD_Command
    FOR idx2 = 0 TO (scrWidth - 1)
      READ (eeAddr + idx2), char
      IF (char = CR) THEN EXIT
      GOSUB LCD_Write_Char
    NEXT
    IF (char = CR) THEN EXIT
    eeAddr = eeAddr + 1
    PAUSE LcdScrollTm
  LOOP
  RETURN
```

This routine requires three variables to be setup before calling: crsrPos points to the first (left-most) position of our scrolling window, scrWidth is the width of the scrolling window, and eeAddr points to the (CR-terminated) string we'll print. This string is stored in a DATA statement.

This code will run in a continuous loop until it his the end of the string. The first step is to position the LCD cursor at the leftmost window position. Next, the code will loop through the number of characters to print, pulling a character from the EEPROM and putting in the LCD. This works because the LCD initialization causes the cursor to advance to the right after each character print.

After printing the string segment, the eeAddr pointer is incremented and a short delay is inserted so that we can actually read the characters. When we increment the eeAddr pointer, what we're actually doing is sliding the [virtual] display window over the string. The process continues until a CR is encountered. This will cause the FOR-NEXT print loop to exit, then immediately break out of the outer DO-LOOP.

This routine is used to print a program name banner at the beginning. Once that's done, we go right into the hear to the program. The main section is a bit long, but we'll break it up into small chunks so it shouldn't be too tough to follow:

```
Main:
  char = LcdHome
  GOSUB LCD_Command
  char = LcdLine2
  GOSUB LCD_Command
  eeAddr = Controls
  GOSUB LCD_Put_String
```

We start at the top of the main program loop by making sure the LCD display is aligned by using the Home command. Then we'll move to the second line and print a controls string for the input buttons. When you download the main listing you'll see that I've created a couple custom characters for use in the LCD. The character codes for the custom characters are embedded in the Controls string.

```
Check_Level:
  IF (mnuLevel = 0) THEN
    GOSUB Show_Pgm
  ELSE
    GOSUB Show_Freq
  ENDIF
```

Moving on we'll check the current menu level (0 for selecting program pattern, 1 for setting frequency) and update the first line of the display accordingly.

```
Show_Pgm:
  char = LcdHome
  GOSUB LCD_Command
  LOOKUP pgm, [Pgm1, Pgm2, Pgm3, Pgm4], eeAddr
  GOSUB LCD_Put_String
  RETURN
```

The Show_Pgm subroutine uses a LOOKUP table to convert the program value (0 to n) into an EEPROM pointer address. This addressed is passed to LCD_Put_String and the CR-terminated string is printed on the LCD at the current character position.

```
Show_Freq:
  char = LcdHome
  GOSUB LCD_Command
  eeAddr = FrMsg
  GOSUB LCD_Put_String
  char = LcdLine1 + 4
  GOSUB LCD_Command

Write_Freq_Value:
  IF (freq < 10) THEN
    char = " "
  ELSE
    char = (freq DIG 1) + "0"
  ENDIF
  GOSUB Lcd_Write_Char
  char = (freq DIG 1) + "0"
  GOSUB Lcd_Write_Char
  RETURN
```

The Show_Freq subroutine is similar, but has a bit more work to do so we've split it into two sections (entry points). The first section moves the cursor to the start of Line 1 and prints the string from EEPROM. The cursor is repositioned to the place where the frequency value is printed and the second section of code handles that. For neatness, we're going the extra mile of space-padding single-digit values. It just makes the display look more professional.

With the display updated, the final step is to wait for a button press and process the input.

```
  DO
    GOSUB LCD_Get_Buttons
  LOOP UNTIL (buttons > %0000)
```

This code is as simple as it looks – it scans the buttons and stays in this loop until one is pressed. Let's look at the button scan routine.

```
LCD_Get_Buttons:
  BusDirs = %0000
  buttons = %1111
  FOR idx2 = 1 TO 10
    buttons = buttons & BusIns
    PAUSE 5
  NEXT
```

```
  BusDirs = %1111
  RETURN
```

This code probably looks familiar to some of you as we have used it in the past to debounce multiple input pins. The key here is that we have to make the bus pins inputs on entry to this routine, then reset them to outputs before we leave. The reason for this is that the program will spend more time writing to the LCD than reading the buttons, so it's best to take care of the bus setup here than burden all of the LCD code with it.

For those who haven't seen this in the past, it's very easy. The routine assumes the buttons are pressed. It reads them and ANDs the current inputs with the buttons value. If a button isn't pressed (at all or due to "bounce") then the input will be zero and that ANDed with the buttons value will clear the input for this routine. What this means, then, is that a button must be pressed on entry to this routine and for the duration of it to register as a valid button press.

Before writing the program I decided that only one button at a time would be looked at and in a specific order. If a button was pressed, it would be processed and the rest ignored on this particular scan. The first thing that is checked is the "Run" button. If pressed, it will launch the LEDs using the current program selection and flash frequency.

```
IF (btnD = Pressed) THEN Run_Program
```

Nothing tricky here – just reiterates how cleanly variable aliases and program constants can make our code.

As it turns out, there are two types of programs: flashers and sequencers. Flashers basically have two states, sequencers have three or more patterns that will run in sequence on the LEDs.

```
Run_Program:
  WRITE LastPgm, pgm
  WRITE LastFreq, freq

  char = LcdCls
  GOSUB LCD_Command
  GOSUB Show_Pgm
  char = LcdLine2 + 4
  GOSUB LCD_Command
  GOSUB Write_Freq_Value
  eeAddr = FrMsg + 6
  GOSUB LCD_Put_String
```

This first section saves our current program and frequency settings to EEPROM so that they'll be loaded the next time we start the program. It's a nice feature: to have a device remember what it did last. After the settings are saved, the LCD is cleared and updated with the settings. The program name is printed on the first line, the frequency is written on the second and right-justified. Here's where we can see why there were two entry points in the Show_Freq subroutine. The second entry point allows us to write the frequency at the current cursor position. Notice that the last eeAddr setting is "FrmMsg + 6." What this is doing is pointing to the "Hz" characters in the frequency label string.

```
  BusDirs = %0000
  DO WHILE (BusIns > %0000) : LOOP
```

This next step makes the bus pins inputs and waits for the Run button to be released. We need to do this since a button press will be used to stop the program cycle. That being the case, we must make sure the buttons are cleared before we even start running.

```
  period = 1000 / freq
  LOOKUP pgm, [M0, M1, M2, M3], eeAddr
  READ eeAddr, LEDs
```

Since we're getting close to actually launching the program, we calculate the period (in milliseconds) that will be used for LED timers. We'll also look up the start of the program pattern and place it on the LEDs.

```
  IF (pgm < 3) THEN
    timer1 = period / 2                         ' on time
    timer2 = period - timer1                    ' off time
    GOSUB Two_State
  ELSE
    states = 6                                  ' eight states in chase
    timer1 = period / states                    ' divide period
    GOSUB Multi_State
  ENDIF
```

Finally, we check to see if the selected program runs in two-state or multi-state mode. For this program I did it with an IF-THEN construct. If I ever expand this to more modes, what I'd probably do is embed the number of states in the DATA table ahead of the patterns to keep future edits to a minimum.

Two state programs do just that: the LEDs start in one state, then switch to the opposite. Each state will be on for one-half the set period. The first timer is calculated by dividing the period into 1000 (so we get milliseconds for PAUSE out), the second by subtracting the first timer from the period. This will prevent rounding errors, though I don't think that they'd damage the effectiveness of this particular application.

Okay, here's the two-state code:

```
Two_State:
  DO WHILE (BusIns = %0000)
    PAUSE timer1
    LEDs = ~LEDs
    PAUSE timer2
    LEDs = ~LEDs
  LOOP
  DO WHILE (BusIns > %0000) : LOOP
  LEDs = %00000000
  RETURN
```

As you can see, this subroutine runs in a continuous loop until one of the buttons is pressed. The active portion holds the LEDs in their preset state for the duration of `timer1`, then inverts the state of the LEDs. The second state is held for the duration of `timer2` and then the LEDs are inverted again (back to their original state).

When a button is pressed, the main loop will be terminated. A second loop will hold the subroutine until all buttons are released. This will make sure that the button press to stop the current cycle doesn't interfere with our menu processing. The last step is to clear the LEDs and return to the main body.

```
Multi_State:
  idx1 = 0
  DO WHILE (BusIns= %0000)
    READ (eeAddr + idx1), LEDs
    PAUSE timer1
    idx1 = idx1 + 1 // states
  LOOP
  DO WHILE (BusIns > %0000) : LOOP
  LEDs = %00000000
  RETURN
```

The multi-state code also runs in a loop. At its heart it reads a state value from the pattern table and holds them on the LEDs for a duration that is the period divided by the number of states. As with the two-state mode, this code will run until a button is pressed, then will force a button release. When we do return to the main loop, the bus is returned to outputs for the LEDs and the program cycled back to the top of main to get refresh the display and wait for another button input.

Okay, we've run a program, but we haven't discussed changing menu levels or setting our parameters. Let's look at those and wrap up.

```
  IF (btnC = Pressed) THEN
    mnuLevel = mnuLevel + 1 // NumLevels
    GOTO Clear_Buttons
  ENDIF
```

Button C on the display is used to change menu levels. This application only has two levels (program and frequency), but the code is written to accommodate more. The way it's structured it will advance to the next level, wrapping around to zero (via the modulus operator) when we're at the end. With the new level set, the other buttons are cleared and Main loop started again.

Buttons A and B are used to increase or decrease the program selection or frequency setting. Here's the code for both:

```
  IF (btnA = Pressed) THEN
    SELECT mnuLevel
      CASE 0
        pgm = pgm + 1 // NumPgms
      CASE 1
        freq = freq // 20 + 1
    ENDSELECT
    GOTO Clear_Buttons
  ENDIF

  IF (btnB = Pressed) THEN
    SELECT mnuLevel
      CASE 0
        pgm = pgm + (NumPgms - 1) // NumPgms
      CASE 1
        freq = freq - 1
        IF (freq = 0) THEN freq = 20
    ENDSELECT
    GOTO Clear_Buttons
  ENDIF
```

What we're doing is using a `SELECT-CASE` structure to determine whether we increase/decrease the program number or frequency value. Again, we do this so we can add more features later without major reconstructive surgery (as if we had used `IF-THEN` to check the menu level). The `CASE` code is very easy. The one thing to note is that there is no clever way to use the modulus operator for an automatic wrap from 20 to 1 as we need for the frequency setting. Yes, we could have used 0 to 19 and dealt with it internally, but this probably would have lead to confusion and then not been worth doing.

Okay, that's it. Funny, how a seemingly "simply" application on the outside can be moderately complex on the inside. Keep this in mind when you're planning future projects. You know the old saying, "It's harder than it looks."

Oh, the little girl in the photo is my wildly cute niece Marissa testing the hypno-goggles created by her favorite "Uncle Jonny." The effect? Lots of giggles, but no out-of-body or other parapsychological experiences.

A Reminder

For those of you who may be new to Nuts & Volts or just recently discovered "Stamp Applications" you may be wondering why I seemingly skipped over a lot of details regarding the LCD. The reason is that between Scott Edwards and me, we've covered a lot of LCD ground in this column. "But I didn't know about Nuts & Volts back then....", you protest. No problem. You can read all the previous issues of this column from the comfort of your own PC. Nuts & Volts has very generously allowed Parallax to republish back-issues as PDF files, and if you want them for you desk you can get them in printed volumes as well.

You can find the back issues on the Parallax web site under Downloads \ Nuts & Volts Columns.

What's Next?

I did this project because I'd been asked to do a menu-based LCD project – the hypno-goggles element just fit into a current personal interest. I have also been getting a lot of requests for more GPS related material. So that's what we'll do next month. We'll look at a GPS-based digital dashboard that I actually "test drove" on my vacation drive from Dallas to Columbus, Ohio. With directions from the Internet and the mileage readout from my project, I was able to drive from my home to my brother's without any hiccups. I'll show you how next time.

Until then, Happy Stamping.

Figure 102.4: Marissa Puts the Hypno-Goggles to Use

```
' =========================================================================
'
'   File....... Mind_Sync.BS2
'   Purpose.... Brainwaive "synchronizer"
'   Author..... Jon Williams
'   E-mail..... jwilliams@parallax.com
'   Started....
'   Updated.... 22 AUG 2002
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =========================================================================


' -----[ Program Description ]---------------------------------------------
'
' This program uses the Parallax LCD AppMod to create a user interface for
' controlling the flash rate and pattern of LEDs.  The LEDs, mounted in a
' hood or glasses, shine into the eyes (through the lids) in order to
' synchronize the brainwaves to the flash rate.
'
' This program uses PBASIC 2.5 features so that it can run on any BASIC
' Stamp 2 series without modification.
'
' =========
'  NOTICE!
' =========
'
' This software is provided for EDUCATIONAL PURPOSES only.  No guarantee of
' suitability for any health/medical application is expressed or implied.


' -----[ I/O Definitions ]-------------------------------------------------

E               PIN     1                       ' LCD Enable (1 = enabled)
RW              PIN     2                       ' Read/Write\
RS              PIN     3                       ' Reg Select (1 = char)
BusDirs         VAR     DIRB                    ' dirs for I/O redirection
BusOuts         VAR     OUTB                    ' for output to LCD
BusIns          VAR     INB                     ' input from LCD/buttons
LEDs            VAR     OUTH                    ' LEDs on P8 - P15


' -----[ Constants ]-------------------------------------------------------

#DEFINE _LcdReady = ($STAMP = BS2P) OR ($STAMP = BS2PE)

LcdCls          CON     $01                     ' clear the LCD
LcdHome         CON     $02                     ' move cursor home
LcdCrsrL        CON     $10                     ' move cursor left
```

```
LcdCrsrR        CON     $14                     ' move cursor right
LcdDispL        CON     $18                     ' shift chars left
LcdDispR        CON     $1C                     ' shift chars right
LcdDDRam        CON     $80                     ' Display Data RAM control
LcdCGRam        CON     $40                     ' Custom character RAM
LcdLine1        CON     $80                     ' DDRAM address of line 1
LcdLine2        CON     $C0                     ' DDRAM address of line 2

LcdWidth        CON     8                       ' chars in LCD line
LcdScrollTm     CON     250                     ' LCD scroll timing (ms)

NumLevels       CON     2                       ' program and freq
NumPgms         CON     4                       ' number of programs

Pressed         CON     1                       ' buttons are active high


' -----[ Variables ]----------------------------------------------------

eeAddr          VAR     Word                    ' ee address pointer
crsrPos         VAR     Byte                    ' cursor position
scrWidth        VAR     Nib                     ' scroll window width
char            VAR     Byte                    ' character sent to LCD
idx1            VAR     Byte                    ' loop counter
idx2            VAR     Byte                    ' loop counter

buttons         VAR     Nib
btnA            VAR     buttons.BIT0            ' left-most button
btnB            VAR     buttons.BIT1
btnC            VAR     buttons.BIT2
btnD            VAR     buttons.BIT3            ' right-most

pgm             VAR     Nib                     ' program selection
states          VAR     Nib                     ' LED states (for tables)
freq            VAR     Byte                    ' frequency (1 to 20 Hz)
period          VAR     Word                    ' period of set freq
timer1          VAR     Word                    ' LED timers
timer2          VAR     Word

mnuLevel        VAR     Nib                     ' menu level


' -----[ EEPROM Data ]--------------------------------------------------

LastPgm         DATA    0                       ' last pgm run
LastFreq        DATA    10                      ' last freq setting

Banner          DATA    "        BRAINWAVE S"
                DATA    "YNCHRONIZER        ", CR

Pgm1            DATA    "Flash   ", CR
```

```
Pgm2            DATA    "Alt1    ", CR
Pgm3            DATA    "Alt2    ", CR
Pgm4            DATA    "Chase   ", CR
FrMsg           DATA    "Fr    Hz", CR

Controls        DATA    0, " ", 1, "  ", 2, " ", 3, CR

' customer characters
'
UpArrow         DATA    $04, $0E, $15, $04, $04, $04, $04, $00
DnArrow         DATA    $04, $04, $04, $04, $15, $0E, $04, $00
Enter           DATA    $00, $01, $05, $09, $1F, $08, $04, $00
RunBtn          DATA    $00, $0E, $1F, $1B, $1F, $0E, $00, $00

' mode patterns
'
M0              DATA    %11111111
M1              DATA    %11001100
M2              DATA    %00001111
M3              DATA    %00000001, %00010000, %10000000
                DATA    %00100000, %00000010, %00000100


' -----[ Initialization ]--------------------------------------------------

Initialize:
  DIRL = %11111110                              ' setup pins for LCD
  DIRH = %11111111                              ' LED pins are outputs

LCD_Init:
  PAUSE 500                                     ' let the LCD settle
  #IF (_LcdReady) #THEN
    LCDCMD E, %00110000 : PAUSE 5               ' 8-bit mode
    LCDCMD E, %00110000 : PAUSE 0
    LCDCMD E, %00110000 : PAUSE 0
    LCDCMD E, %00100000 : PAUSE 0               ' 4-bit mode
    LCDCMD E, %00101000 : PAUSE 0               ' 2-line mode
    LCDCMD E, %00001100 : PAUSE 0               ' no crsr, no blink
    LCDCMD E, %00000110                         ' inc crsr, no disp shift
  #ELSE
    BusOuts = %0011                             ' 8-bit mode
    PULSOUT E, 3 : PAUSE 5
    PULSOUT E, 3 : PAUSE 0
    PULSOUT E, 3 : PAUSE 0
    BusOuts = %0010                             ' 4-bit mode
    PULSOUT E, 3
    char = %00101000                            ' 2-line mode
    GOSUB LCD_Command
    char = %00001100                            ' no crsr, no blink
    GOSUB LCD_Command
    char = %00000110                            ' inc crsr, no disp shift
```

```
    GOSUB LCD_Command
  #ENDIF

Download_Chars:                                 ' download custom chars
  char = LcdCGRam                               ' point to CG RAM
  GOSUB LCD_Command                             ' prepare to write CG data
  FOR idx1 = UpArrow TO (RunBtn + 7)            ' build 4 custom chars
    READ idx1, char                             ' get byte from EEPROM
    GOSUB LCD_Write_Char                        ' put into LCD CG RAM
  NEXT

Intro:
  char = LcdCls                                 ' clear the LCD
  GOSUB LCD_Command
  PAUSE 500
  crsrPos = LcdLine1                            ' start of scroll window
  scrWidth = LcdWidth                           ' use full width of LCD
  eeAddr = Banner                               ' point to message
  GOSUB LCD_Scroll_String                       ' scroll it

Setup:
  READ LastPgm, pgm                             ' load last settings
  READ LastFreq, freq


' -----[ Program Code ]----------------------------------------------------

Main:
  char = LcdHome
  GOSUB LCD_Command
  char = LcdLine2                               ' move to line 2
  GOSUB LCD_Command
  eeAddr = Controls
  GOSUB LCD_Put_String                          ' display controls

Check_Level:
  IF (mnuLevel = 0) THEN                        ' show pgm mode or freq
    GOSUB Show_Pgm
  ELSE
    GOSUB Show_Freq
  ENDIF

  DO
    GOSUB LCD_Get_Buttons                       ' scan buttons
  LOOP UNTIL (buttons > %0000)                  ' until some input

  ' Process input
  ' -- only one button is allowed
  ' -- others are discarded

  IF (btnD = Pressed) THEN Run_Program
```

```
  IF (btnC = Pressed) THEN
    mnuLevel = mnuLevel + 1 // NumLevels        ' next menu level
    GOTO Clear_Buttons
  ENDIF

  IF (btnA = Pressed) THEN
    SELECT mnuLevel
      CASE 0
        pgm = pgm + 1 // NumPgms                ' next program
      CASE 1
        freq = freq // 20 + 1                   ' increase freq
    ENDSELECT
    GOTO Clear_Buttons
  ENDIF

  IF (btnB = Pressed) THEN
    SELECT mnuLevel
      CASE 0
        pgm = pgm + (NumPgms - 1) // NumPgms    ' previous program
      CASE 1
        freq = freq - 1                         ' decrease freq
        IF (freq = 0) THEN freq = 20            ' rollunder
    ENDSELECT
    GOTO Clear_Buttons
  ENDIF

Clear_Buttons:
  buttons = %0000
  PAUSE 250                                     ' auto-repeat delay
  GOTO Check_Level

Run_Program:
  WRITE LastPgm, pgm                            ' save settings
  WRITE LastFreq, freq

  char = LcdCls
  GOSUB LCD_Command
  GOSUB Show_Pgm                                ' display program
  char = LcdLine2 + 4
  GOSUB LCD_Command
  GOSUB Write_Freq_Value                        ' write freq to display
  eeAddr = FrMsg + 6                            ' point to "Hz"
  GOSUB LCD_Put_String                          ' show frequency

  BusDirs = %0000                               ' give bus to buttons
  DO WHILE (BusIns > %0000) : LOOP              ' force button release

  ' prep timers and run
  period = 1000 / freq                          ' program period
  LOOKUP pgm, [M0, M1, M2, M3], eeAddr          ' point to pattern
```

```
  READ eeAddr, LEDs                                ' load first pattern

  IF (pgm < 3) THEN
    timer1 = period / 2                            ' on time
    timer2 = period - timer1                       ' off time
    GOSUB Two_State
  ELSE
    states = 6                                     ' eight states in chase
    timer1 = period / states                       ' divide period
    GOSUB Multi_State
  ENDIF
  BusDirs = %1111                                  ' return bus to LCD
  GOTO Main

  END


' -----[ Subroutines ]-----------------------------------------------------

' Display name of current program

Show_Pgm:
  char = LcdHome
  GOSUB LCD_Command
  LOOKUP pgm, [Pgm1, Pgm2, Pgm3, Pgm4], eeAddr  ' get string address
  GOSUB LCD_Put_String                           ' print string
  RETURN


' Display current frequency

Show_Freq:                                       ' write string from ee
  char = LcdHome
  GOSUB LCD_Command
  eeAddr = FrMsg
  GOSUB LCD_Put_String
  char = LcdLine1 + 4                            ' move to 10's position
  GOSUB LCD_Command

Write_Freq_Value:                                ' second entry - write value
  IF (freq < 10) THEN
    char = " "
  ELSE
    char = (freq DIG 1) + "0"                    ' write 10's digit
  ENDIF
  GOSUB Lcd_Write_Char
  char = (freq DIG 0) + "0"                      ' write 1's digit
  GOSUB Lcd_Write_Char
  RETURN
```

```
' This routine is used for patterns where LEDs are in one of
' two states and the states can be swapped by XORint the LED
' values with %11111111

Two_State:
  DO WHILE (BusIns = %0000)                        ' run until button press
    PAUSE timer1
    LEDs = ~LEDs                                   ' invert
    PAUSE timer2
    LEDs = ~LEDs
  LOOP
  DO WHILE (BusIns > %0000) : LOOP                 ' force button release
  LEDs = %00000000
  RETURN


' This state is used to read specific LED patterns
' from an EEPROM table.
' -- starting pattern addres in 'addr'
' -- number of patterns in 'states'

Multi_State:
  idx1 = 0                                         ' current state pointer
  DO WHILE (BusIns= %0000)                         ' run until button press
    READ (eeAddr + idx1), LEDs                     ' get pattern
    PAUSE timer1
    idx1 = idx1 + 1 // states                      ' point to next
  LOOP
  DO WHILE (BusIns > %0000) : LOOP                 ' force button release
  LEDs = %00000000
  RETURN


' -----[ LCD Subroutines ]-------------------------------------------------

' Writes stored (in DATA statement) CR-terminated string to LCD
' -- position LCD cursor
' -- point to CR-terminated string (first location in 'eeAddr')

LCD_Put_String:
  DO
    READ eeAddr, char
    IF (char = CR) THEN EXIT
    GOSUB LCD_Write_Char
    eeAddr = eeAddr + 1
  LOOP
  RETURN


' Scroll a message across LCD line
' -- set starting position in 'crsrPos'
```

```
' -- set width of scrolling window in 'scrWidth'
' -- point to 0-terminated string (first location in 'eeAddr')
' -- strings should be padded with scrWidth spaces on each end

LCD_Scroll_String:
  DO
    char = crsrPos                              ' move to start of window
    GOSUB LCD_Command
    FOR idx2 = 0 TO (scrWidth - 1)              ' write chars in window
      READ (eeAddr + idx2), char
      IF (char = CR) THEN EXIT                  ' stop if end of string
      GOSUB LCD_Write_Char
    NEXT
    IF (char = CR) THEN EXIT
    eeAddr = eeAddr + 1                         ' scroll
    PAUSE LcdScrollTm
  LOOP
  RETURN


' Send command to LCD
' -- put command byte in 'char'

LCD_Command:                                    ' write command to LCD
  #IF (_LcdReady) #THEN
    LCDCMD E, char
    RETURN
  #ELSE
    LOW RS
    GOTO LCD_Write_Char
  #ENDIF


' Write character to current cursor position
' -- but byte to write in 'char'

LCD_Write_Char:                                 ' write character to LCD
  #IF (_LcdReady) #THEN
    LCDOUT E, 0, [char]
  #ELSE
    BusOuts = char.HIGHNIB                      ' output high nibble
    PULSOUT E, 3                                ' strobe the Enable line
    BusOuts= char.LOWNIB                        ' output low nibble
    PULSOUT E, 3
    HIGH RS                                     ' return to character mode
  #ENDIF
  RETURN


' Read and debounce the LCD AppMod buttons
```

```
LCD_Get_Buttons:
  BusDirs = %0000                                  ' make bus inputs
  buttons = %1111                                  ' assume all pressed
  FOR idx2 = 1 TO 10
    buttons = buttons & BusIns                     ' make sure button held
    PAUSE 5                                        ' debounce 10 x 5 ms
  NEXT
  BusDirs = %1111                                  ' return bus to outputs
  RETURN
```