



Column #98 June 2003 by Jon Williams:

Color Me Tickled

I've been around BASIC Stamps and other microcontrollers long enough that I really should be jaded, but from time-to-time a really neat device comes along. Neat new devices are like great golf games: one good one can make you forget all the bad or boring ones for the past several months.

Back in December I was in our California office when a package arrived from a company called Bueno Systems. Inside was a bunch of plastic pieces that I dutifully assembled into what turned out to be an M&M® candy sorter. You can see a picture of this cool little machine on page 63 of the new Parallax catalog. This thing is way beyond entertaining as it loads an M&M into a scanner, "looks" at it, and then sorts it into a specific bin for that color.

At the heart of this device – and the subject of this month's article – is the TCS230 color sensor from TAOS (Texas Advanced Optoelectronics Solutions). The TCS230 is similar to the TSL230 that both Scott Edwards and I have discussed in this column in the past. It is a light-to-frequency converter. This difference is that the TCS230 uses an array of photo detectors; some have red filters, some green, some blue, and the rest have no filters. Using two pins, we can select which set of detectors are enabled. What this lets us do is "see" a particular primary color (or overall level if no filter selected).

In case you forgot your high school science lessons (or you haven't been through them yet!) let's take just a second to talk about light color theory. When all the frequencies of visible light are mixed together we get white. The three primary constituents of white light are red, green and blue. By mixing red, green and blue in varying proportions we can create any color. A great example of this RGB color mixing is your television. In fact, if you get very close to the screen, you can actually see the individual red, green and blue pixels that make up each color point on a TV scan line.

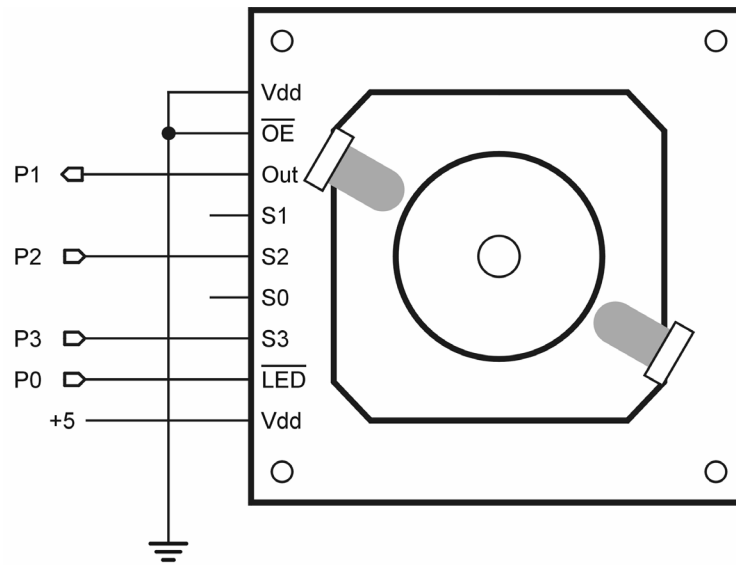
Getting back to the TCS230 ... it can't actually "see" color, it simply measures light intensity. What happens is that a given filter only allows that particular primary color to pass through, blocking the others. When we select the red filter, for example, only red light passes through so we can measure the intensity of the red light that is falling onto the TCS230. By repeating this process for green and blue as well, we're able to analyze the color that the TCS230 "sees." The color is expressed with three values, R-G-B, as we use with televisions and computer monitors.

It turns out that the folks at TAOS are also fans of BASIC Stamps so they've created an application kit for the TCS230. The kit includes the sensor mounted on a PCB with a prefocused lens, clear LEDs for illumination, and a separate adapter board that plugs into the AppMod connector on the Parallax Board-of-Education. The adaptor allows the BASIC Stamp to control two sensor PCB assemblies.

For simplicity, we're going to connect directly to the sensor board. This will let us decide which pins we want to use to control and read the TCS230 – and we can use fewer pins. Using this minimal configuration, we need just four Stamp I/O pins to select the color filter, enable or disable the LEDs, and to read the frequency output of the TCS230.

Figure 98.1 shows the connections for our code. There are pull-ups on the PCB so the pins that are not connected actually get pulled high. Of note are pins labeled S0 and S1 which are used to divide the frequency output of the TCS230. Left unconnected, the output frequency is not divided. It's important to point out that in very bright conditions the output frequency of the TCS230 can exceed the capabilities of the BS2's COUNT function. In my experiments in "typical" ambient light, I haven't found this to be a problem. The sensor PCB does have a couple of open jumper connections so you can hard-wire the divider selection if you decide you need to do that, and don't want to control the divider from the BASIC Stamp.

Okay, let's get to the code and see how we can use the TCS230 to scan and identify specific colors. Now, even if you're not interested in color identification, you may want to stick with me here because there are a couple neat tricks in this program that have applicability in a lot of projects. Okay? Let's go.

Figure 98.1: TCS230 Color Evaluation Module Connected to BASIC Stamp**Workin' Our Plan**

Since I did my "Plan your work, work you plan" rant a couple months ago, let's start with our projects goals:

1. Calibrate the sensor for white (white balance)
2. Scan and store known color samples
3. Scan and identify unknown samples

If you've never operated a video camera, you may not have come across the term "white balance." What this is, essentially, is telling the system, "This is white." In theory, white light is composed of equal amounts of red, green and blue light, but the truth is that the light sensors in the TCS230 (and our video cameras) are not equally sensitive to the various constituents. So what we do is put a white target in front the sensor, illuminate it, measure the red, green and blue levels, then create scaling factors so that the resultant levels for each color when it "sees" white are equalized. What the scaling factors do is account for the variances in sensitivity between the constituent color sensors, as well as any color bias from the illumination source.

How Green is Green?

Before we can scale our color readings, we have to take them, so let's start with an essential element of the program: reading data from the sensor. Reading any of the constituent colors from the TCS230 is very simple: We select the color filter, turn on the LEDs to illuminate the target, measure the output using COUNT, then turn off the LEDs. Here's the code:

```
Read Color:
  SELECT filter
    CASE Red
      LOW TcsS2
      LOW TcsS3

    CASE Green
      HIGH TcsS2
      HIGH TcsS3

    CASE Blue
      LOW TcsS2
      HIGH TcsS3

    CASE ELSE
      HIGH TcsS2
      LOW TcsS3
  ENDSELECT

  TcsLeds = IsOn
  COUNT TcsFreq, ScanTime, rawColor
  TcsLeds = IsOff
  RETURN
```

For this routine we will pass the constituent color selection in the variable filter. A SELECT-CASE structure takes care of setting the filter control output pins. As you can see, the actual code is shorter than the explanation. The program uses constants for the pin numbers as well as the scan time used by the count function. Based on the TAOS examples and some experimentation, the scan time in this program is 10 milliseconds. Note that no delay is required between enabling the LEDs and using the COUNT function. LEDs are "instant on" devices and don't require any warm-up like incandescent and other light sources.

The value returned by this subroutine is a word variable called rawColor. Remember this isn't scaled for color sensitivity or illumination color bias.

Fractions On The Fly

Beyond the pure "neato" factor of this program, one of the things I like best is the ability to calculate a fractional scaling factor as required for white balancing the sensor. "Fractions?" you wonder ... "The Stamp doesn't do fractions." Well, yes and no. True, the Stamp doesn't do floating-point math, but we can multiply by a fractional value using the `*/` (star-slash) and `**` (star-star) operators. We're going to use `*/` here because it allows values greater than one.

How do we do it? Let's say, for example, that we measure a level of 85 and would like to scale that level to 100. We can determine the scale factor with simple math:

$$\text{factor} = \text{target} \div \text{measurement}$$

Using the numbers above, we'd end up with a scaling factor of 1.176. To get this into a format that can be used by the `*/` operator we have to multiply the factor by 256. Since the math we're doing is straight division and multiplication, we can actually rearrange the order a bit to make it BASIC Stamp friendly and eliminate the integer-math truncation of the fractional part:

$$\text{factor} = \text{target} \times 256 \div \text{measurement}$$

What we'd end up with using 85 as our measurement and 100 as the target is 301. Going back, 301 divided by 256 is 1.175 – pretty close to the 1.176 we calculated earlier.

To white balance the sensor, then, we must place a white target in front of it, read each of the constituent colors and then calculate scaling values for each of them.

```
White Balance:
  filter = Red
  GOSUB Read Color
  calRed = ScaleMax * 256 / rawColor
  filter = Green
  GOSUB Read Color
  calGrn = ScaleMax * 256 / rawColor
  filter = Blue
  GOSUB Read Color
  calBlu = ScaleMax * 256 / rawColor
  RETURN
```

Easy, huh? And yet very useful in this and other projects where we want to scale a [linear] input value to a specified maximum. Okay, now that the sensor knows what white looks like, we need to "teach" it the various colors we want to identify later.

Now that we have scaling factors for the constituents, reading the calibrated RGB colors is a no-brainer:

```
Read_RGB:
  filter = Red
  GOSUB Read_Color
  redVal = rawColor */ calRed MAX ScaleMax
  filter = Green
  GOSUB Read_Color
  grnVal = rawColor */ calGrn MAX ScaleMax
  filter = Blue
  GOSUB Read_Color
  bluVal = rawColor */ calBlu MAX ScaleMax
  RETURN
```

I added in the MAX functions so that slight variations between the ambient light during testing versus white balancing won't cause a roll-over error. In my version of the program the color values are bytes and the ScaleMax value is 100. Using the MAX function is particularly important if you decide to bump ScaleMax to 255 – you certainly don't want your readings rolling over to zero.

When you download the full listing you'll see a color table built into the program and may wonder why we can't just use that. There are a couple really good reasons, actually. You may want to scan different colors, and even if you wanted to scan the same as I did, the lighting in your office will probably be different than in mine (ambient light affects the overall reading). And at the end of the day, anyone who has ever worked in quality control will tell you that you must make sure your test equipment is calibrated before you can use it. So let's calibrate our color scanner.

```
Calibrate Colors:
  FOR colIdx = 0 TO (NumColors - 1)
    DEBUG CLS, "TCS230 Color Calibration: "
    GOSUB Print_Color
    DEBUG CR, CR, "Insert sample. Press a key to scan..."
    TcsLeds = IsOn
    DEBUGIN inKey
    GOSUB Read_RGB
    eePntr = Colors + (3 * colIdx)
    WRITE eePntr, redVal, grnVal, bluVal
  NEXT
  DEBUG CLS
  RETURN
```

The routine will loop through the number of colors set by the NumColors constant (the values are zero-indexed, hence the NumColors – 1 end control value). For each color in the table, we'll see a message screen that looks something like this:

TCS230 Color Calibration: Brown

Insert sample. Press a key to scan...

The color name also comes from a DATA table and I'll explain that in just a moment. The sensor LEDs are lit to help with alignment of small items and after inserting the sample you press a key (read with DEBUGIN), the color gets scanned, then the RGB data is stored in EEPROM. A pointer to the location of the data (the red component) is calculated using the beginning of the table (Colors) and the color index. The values are stored using a multi-byte WRITE statement.

Printing string names is nothing new, but I do want to share a little pointer that can be used to save space in your programs. The string names are stored with zero-terminators like this:

```
' Color Names
CN0      DATA    "Brown", 0
CN1      DATA    "Red", 0
CN2      DATA    "Orange", 0
CN3      DATA    "Yellow", 0
CN4      DATA    "Green", 0
CN5      DATA    "Blue", 0
CN6      DATA    "Violet", 0
```

Okay, here's the code that prints the color names:

```
Print Color:
  LOOKUP colIdx, [CN0, CN1, CN2,
                  CN3, CN4, CN5, CN6], eePntr

Print String:
  DO
    READ eePntr, char
    IF (char = TermChar) THEN EXIT
    DEBUG char
    eePntr = eePntr + 1
  LOOP
  RETURN
```

What I want to point out is that this is two subroutines in one, accomplished by creating two entry labels. The reason for this is that the first entry will LOOKUP the value of eePntr based on the color index, the second section will simply print the string. By doing this we have a specific

routine to print the color name, and a general-purpose routine that will print any string we point to. By "stacking" these routines so that the first falls into the second, we don't have to use GOSUB to call the second from the first; this saves space on the GOSUB stack.

The last bit of hard work is comparing the RGB data table to see if we can match the current scan values. Here's the subroutine that handles the search:

```
Match Color:
  colIdx = 0
  DO WHILE (colIdx < NumColors)
    rgbIdx = 0
    DO WHILE (rgbIdx < 3)
      eePntr = Colors + (colIdx * 3) + rgbIdx
      READ eePntr, testVal
      testVal = ABS(testVal - rgb(rgbIdx))
      IF (testVal > ColorThresh) THEN EXIT
      rgbIdx = rgbIdx + 1
    LOOP
    IF (rgbIdx = 3) THEN EXIT
    colIdx = colIdx + 1
  LOOP
  RETURN
```

Though not too long, it looks a bit complicated. Structurally, there are two loops: the outer loop indexes through the color table, the inner loop indexes through the three RGB components. The inner loop starts by reading the current constituent (R, G or B) from the table then compares it to the constituent of the scan. An array for the scan constituents is created using that aliasing trick I showed you back in April.

redVal	VAR	Byte
grnVal	VAR	Byte
bluVal	VAR	Byte
rgb	VAR	redVal

By aliasing rgb to redVal, we can access the constituent colors as rgb(0) for red, rgb(1) for green, and rgb(2) for blue.

The comparison result ends up in testVal. Notice that we use the ABS operator in case the test value is less than the constituent we're comparing it to. The idea is that we we're looking for an absolute variance, not just in one direction, but in both.

If the variance between the test value and the constituent color is greater than that specified by the ColorThresh constant, the inner loop will be terminated with EXIT and the outer loop will index to the next color. If we do happen to match all three colors, the inner loop will terminate on its own,

and the outer loop will be terminated by a comparison that checks the value of rgbIdx. The value of colIdx holds the match color. If we don't find a match, the outer loop will terminate by itself and the value of colIdx will be the same as NumColors – indicating that no match was found.

The color match routine is the trickiest part of the program so give it a few minutes to sink in. And keep in mind that it is "tunable" with the ColorThresh constant. You can make the routine "looser" by increasing the ColorThresh value, or "tighter" by decreasing it.

Now that the hard work is out of the way, the main program loop is simple:

```
Main:
DO
  GOSUB Read_RGB
  DEBUG "RGB = ",
    DEC3 redVal, ", ",
    DEC3 grnVal, ", ",
    DEC3 bluVal, " "

  GOSUB Match_Color
  IF (colIdx < NumColors) THEN
    GOSUB Print_Color
    DEBUG CR
  ELSE
    DEBUG "No match", CR
  ENDIF

  PAUSE 1000
LOOP
END
```

The main routine is a simple loop that scans the current target, prints the RGB color values then displays the color name if a match was found, otherwise it prints "No match." Figure 98.2 shows what the program display looks like when running (the "No Match" lines occur between samples being placed under the sensor). You can see my setup in the photo. I used a piece of black felt to eliminate reflections from the table and for fun, I decided to scan M&Ms again. M&Ms are convenient since they come in seven colors, they're easy to acquire (you can run to any corner store and pick them up) and when you're done and tired of scanning them you have a treat. You can't beat that. Just be sure to use the "plain" variety as the peanut M&Ms have a tendency to roll around too much!

Have fun with the TCS230 and do buy an extra bag of M&Ms so that you can snack while experimenting. I'll see you next month. Until then, Happy Stamping.

Figure 98.2: Example DEBUG Output with M&Ms

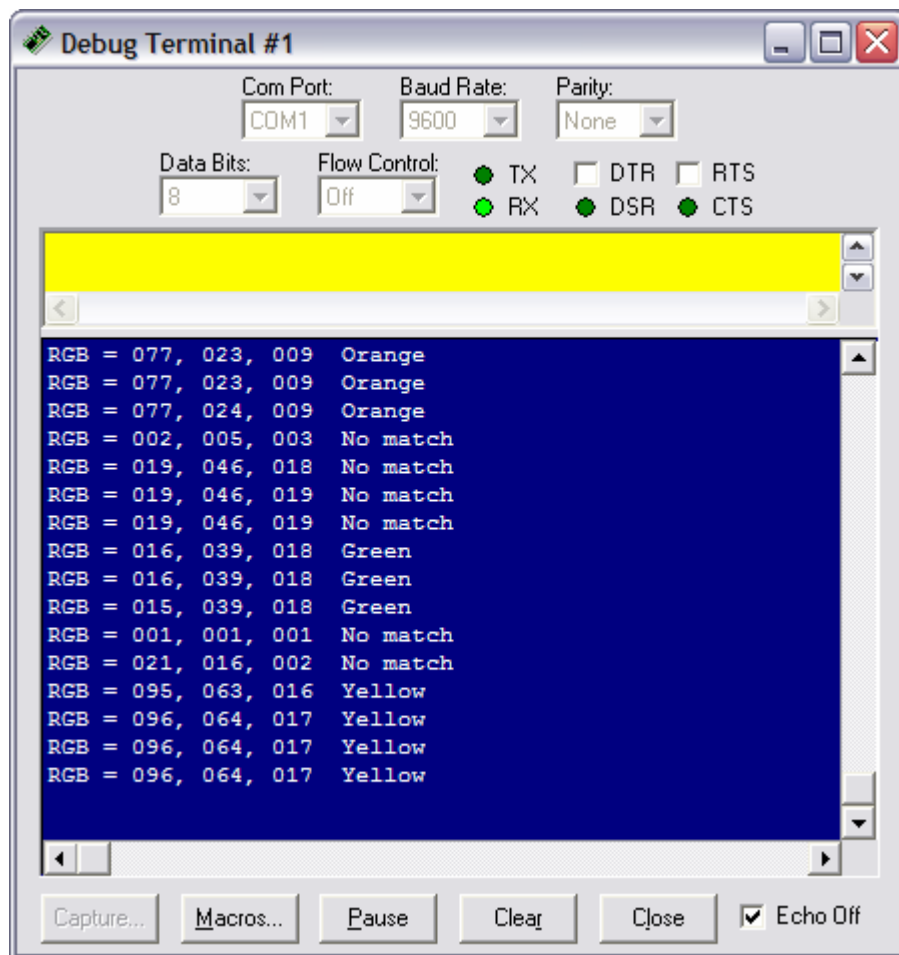
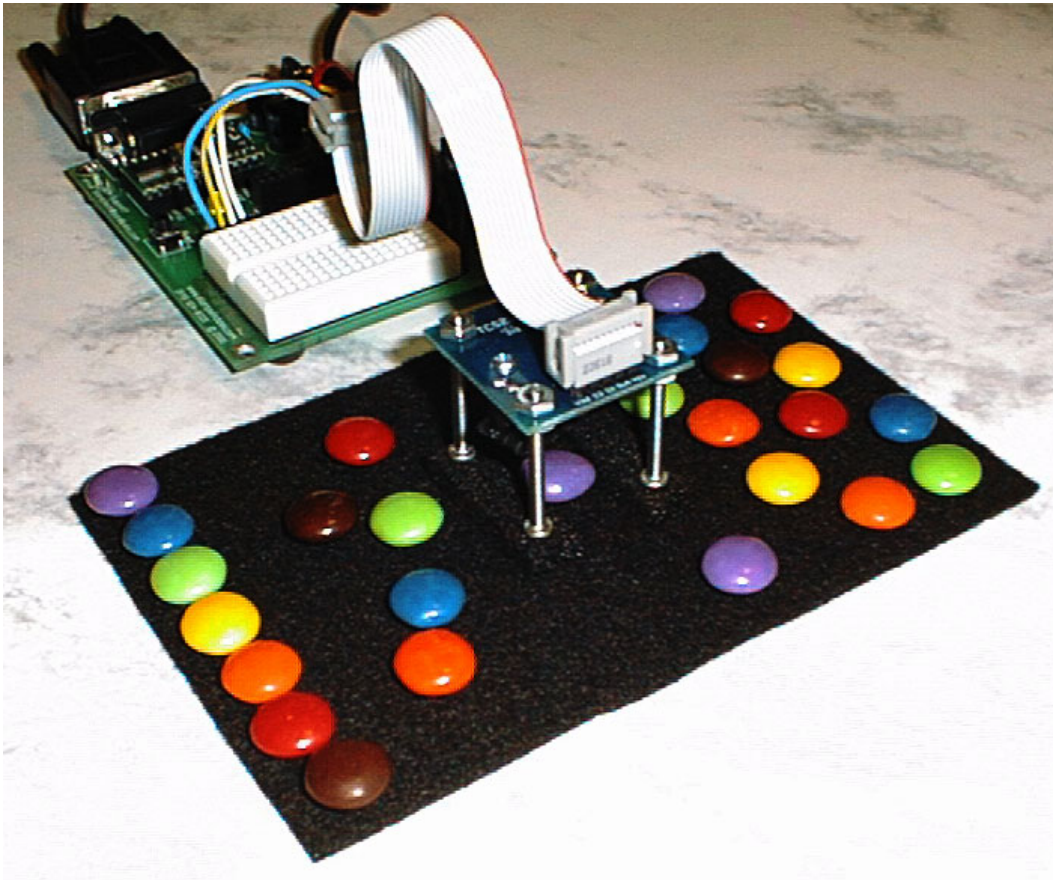


Figure 98.3: Identifying M&M Colors



```

' =====
'
'   File..... Color_Scan.BS2
'   Purpose.... Color Scanner with TAOS TCS230
'   Author..... Jon Williams
'   E-mail..... jwilliams@parallax.com
'   Started....
'   Updated.... 18 APR 2003
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =====

' -----[ Program Description ]-----
'
' Simple color scanner using the TAOS TCS230 color sensor. This program
' uses a direct connection; the TCS dual-sensor adaptor (which is docu-
' mented with the sensor) is not used.
'
' NOTE: OE\ must be tied low to enable TCS230 output and allow control
'       over LEDs.
'
' NOTE: To prevent sensor damage, download the program and run it before
'       connecting the TCS230.

' -----[ Revision History ]-----

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

TcsLeds      PIN    0           ' LEDs enable - active low
TcsFreq      PIN    1           ' freq from TCS230
TcsS2        PIN    2           ' color filter control
TcsS3        PIN    3

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

Red          CON    0           ' TCS230 filter selection
Green        CON    1
Blue         CON    2
Clear        CON    3

ScanTime     CON    10          ' scan time in millisecs
ScaleMax     CON    100         ' max for scaled values

IsOn         CON    0           ' LED control is active low

```

```

IsOff          CON      1

NumColors      CON      7          ' seven M&M colors
ColorThresh    CON      5          ' allowable variance

TermChar       CON      0          ' terminator for strings
StrLen         CON     12          ' max length of names

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

filter         VAR      Nib          ' filter selection
rawColor       VAR      Word         ' raw return from TCS230

calRed         VAR      Word         ' red calibration
calGrn         VAR      Word         ' green calibration
calBlu         VAR      Word         ' blue calibration

redVal        VAR      Byte         ' red value
grnVal        VAR      Byte         ' green value
bluVal        VAR      Byte         ' blue value
rgb           VAR      redVal        ' colors array

inKey          VAR      Byte         ' input from user
colIdx         VAR      Nib          ' color index
rgbIdx         VAR      Nib          ' rgb index
testVal        VAR      Byte         ' test value
eePntr        VAR      Word         ' data table pointer
char           VAR      inKey        ' char to print

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

' RGB data

Colors         DATA    008, 005, 004 ' brown
               DATA    038, 007, 005 ' red
               DATA    075, 022, 008 ' orange
               DATA    086, 060, 011 ' yellow
               DATA    019, 044, 020 ' green
               DATA    005, 009, 023 ' blue
               DATA    021, 017, 031 ' violet

' Color Names

CN0            DATA    "Brown", 0
CN1            DATA    "Red", 0
CN2            DATA    "Orange", 0
CN3            DATA    "Yellow", 0
CN4            DATA    "Green", 0
CN5            DATA    "Blue", 0
CN6            DATA    "Violet", 0

```

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

Setup:
  TcsLeds = IsOff           ' start off
  OUTPUT TcsLeds           ' allow direct control
  GOSUB Calibrate White    ' white balance sensor
  GOSUB Calibrate Colors   ' calibrate color table

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

Main:
  DO
    GOSUB Read RGB         ' scan color
    DEBUG "RGB = ",       ' display components
      DEC3 redVal, " ",
      DEC3 grnVal, " ",
      DEC3 bluVal, " "

    GOSUB Match Color      ' compare scan to table
    IF (colIdx < NumColors) THEN ' match was found
      GOSUB Print_Color
      DEBUG CR
    ELSE
      DEBUG "No match", CR
    ENDIF

    PAUSE 1000             ' delay between scans
  LOOP
END

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

' Calibrates "white" to ambient conditions

Calibrate White:
  DEBUG CLS, "TCS230 White Balance"
  DEBUG CR, CR, "Insert white sample.  Press a key to scan..."
  DEBUGIN inKey
  GOSUB White_Balance
  DEBUG CR, CR, "White balance complete."
  PAUSE 1000
  DEBUG CLS
  RETURN

' Reads "white" and calculates calibration values
```

```

White_Balance:
  filter = Red
  GOSUB Read_Color                                ' read raw red
  calRed = ScaleMax * 256 / rawColor                ' calculate red cal
  filter = Green
  GOSUB Read_Color                                ' read raw green
  calGrn = ScaleMax * 256 / rawColor                ' calculate green scale
  filter = Blue
  GOSUB Read_Color                                ' read raw blue
  calBlu = ScaleMax * 256 / rawColor                ' calculate blue scale
  RETURN

' Calibrates color table to ambient conditions

Calibrate_Colors:
  FOR colIdx = 0 TO (NumColors - 1)                ' loop through all colors
    DEBUG CLS, "TCS230 Color Calibration: "
    GOSUB Print_Color
    DEBUG CR, CR, "Insert sample.  Press a key to scan..."
    TcsLeds = IsOn                                  ' light up scan area
    DEBUGIN inKey
    GOSUB Read_RGB                                  ' scan sample item
    eePntr = Colors + (3 * colIdx)                  ' point to table entry
    WRITE eePntr, redVal, grnVal, bluVal            ' save new data
  NEXT
  DEBUG CLS
  RETURN

' Reads selected color from TCS230
' -- takes "filter" as input
' -- returns "rawColor" as output (unscaled color value)

Read_Color:
  SELECT filter
  CASE Red
    LOW TcsS2
    LOW TcsS3

  CASE Green
    HIGH TcsS2
    HIGH TcsS3

  CASE Blue
    LOW TcsS2
    HIGH TcsS3

  CASE ELSE                                         ' clear -- no filter
    HIGH TcsS2
    LOW TcsS3

```

Column #98: Color Me Tickled

```
ENDSELECT

TcsLeds = IsOn                                ' light sample
COUNT TcsFreq, ScanTime, rawColor           ' return unscaled value
TcsLeds = IsOff
RETURN

' Reads and scales RGB colors

Read RGB:
  filter = Red
  GOSUB Read_Color
  redVal = rawColor */ calRed MAX ScaleMax
  filter = Green
  GOSUB Read_Color
  grnVal = rawColor */ calGrn MAX ScaleMax
  filter = Blue
  GOSUB Read_Color
  bluVal = rawColor */ calBlu MAX ScaleMax
  RETURN

' Compares current color scan with known values in
' table. If match is found, the value of "colIdx"
' will be less than "NumColors"

Match Color:
  colIdx = 0
  DO WHILE (colIdx < NumColors)                ' check known colors
    rgbIdx = 0
    DO WHILE (rgbIdx < 3)                      ' compare rgb components
      eePntr = Colors + (colIdx * 3) + rgbIdx ' point to color table
      READ eePntr, testVal                    ' read known r, g or b
      testVal = ABS(testVal - rgb(rgbIdx))    ' calculate variance
      IF (testVal > ColorThresh) THEN EXIT    ' if out-of-range, next
      rgbIdx = rgbIdx + 1                    ' test next component
    LOOP
    IF (rgbIdx = 3) THEN EXIT                  ' match found
    colIdx = colIdx + 1                       ' try next color
  LOOP
  RETURN

' Print color name
' -- takes "colIdx" as input
' -- allow this to fall through to Print String

Print_Color:
  LOOKUP colIdx, [CN0, CN1, CN2,
                 CN3, CN4, CN5, CN6], eePntr
```



```
' Print a string stored in DATA table
' -- point to first character with "eePntr"

Print String:
DO
  READ eePntr, char
  IF (char = TermChar) THEN EXIT
  DEBUG char
  eePntr = eePntr + 1
LOOP
RETURN
```

' reach character
' end of string?
' no -- print char
' point to next