PARALLAX inc

Web Site: www.parallax.com
Forums: forums.parallax.com
Sales: sales@parallax.com
Technical: support@parallax.com
Office: (916) 624-8333
Fax: (916) 624-8003
Sales: (888) 512-1024
Support: (888) 997-8267

TAOS TEXAS ADVANCED OPTOELECTRONIC SOLUTIONS®

Web Site: www.taosinc.com
Sales: sales@taosinc.com
Office: (972) 673-0759
Fax: (972) 943-0610

TSL1401-DB (#28317): *Linescan Camera Module*

Product Overview

General Description

The TSL1401-DB is a daughterboard that provides a TAOS TSL1401R 128-pixel linear array sensor and a lens. It is designed to plug into a motherboard (e.g. MoBoStamp-pe (p/n 28300), MoBoProp (p/n 28303, in development), Propeller Backpack (p/n 28327)) or the DB-Expander (p/n 28325). This module will allow its host system to "see" in one dimension. Two- dimensional vision can also be achieved by moving either the subject or the sensor in a direction perpendicular to the sensor axis.

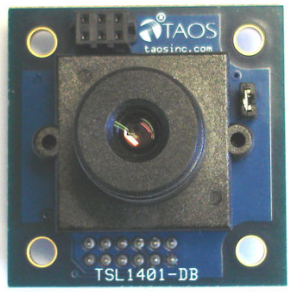
Features

- Provides vision in one dimension with 128-pixel resolution.
- Three-line serial interface with analog intensity output for each pixel.
- Included 7.9mm lens provides a field of view equal to subject distance.
- Plug-compatible with Parallax motherboards.
- Coprocessor driver firmware for the MoBoStamp-pe available for download.
- Can be interfaced directly to a BASIC Stamp for some functions.
- Onboard accessory socket for strobe output or 50/60Hz fluorescent light sync input.
- Runs from 3.3V or 5V supplies. (5V is needed for the optional LED strobe attachment.)

Applications

- Measure height, width, diameter, thickness.
- Locate objects, lines, edges, gaps, holes.
- Count items; measure conveyor coverage.
- Determine volume, shape, orientation.
- Read simple barcodes.
- Learn the principles of machine vision.

What's Included



TSL1401-DB with lens.

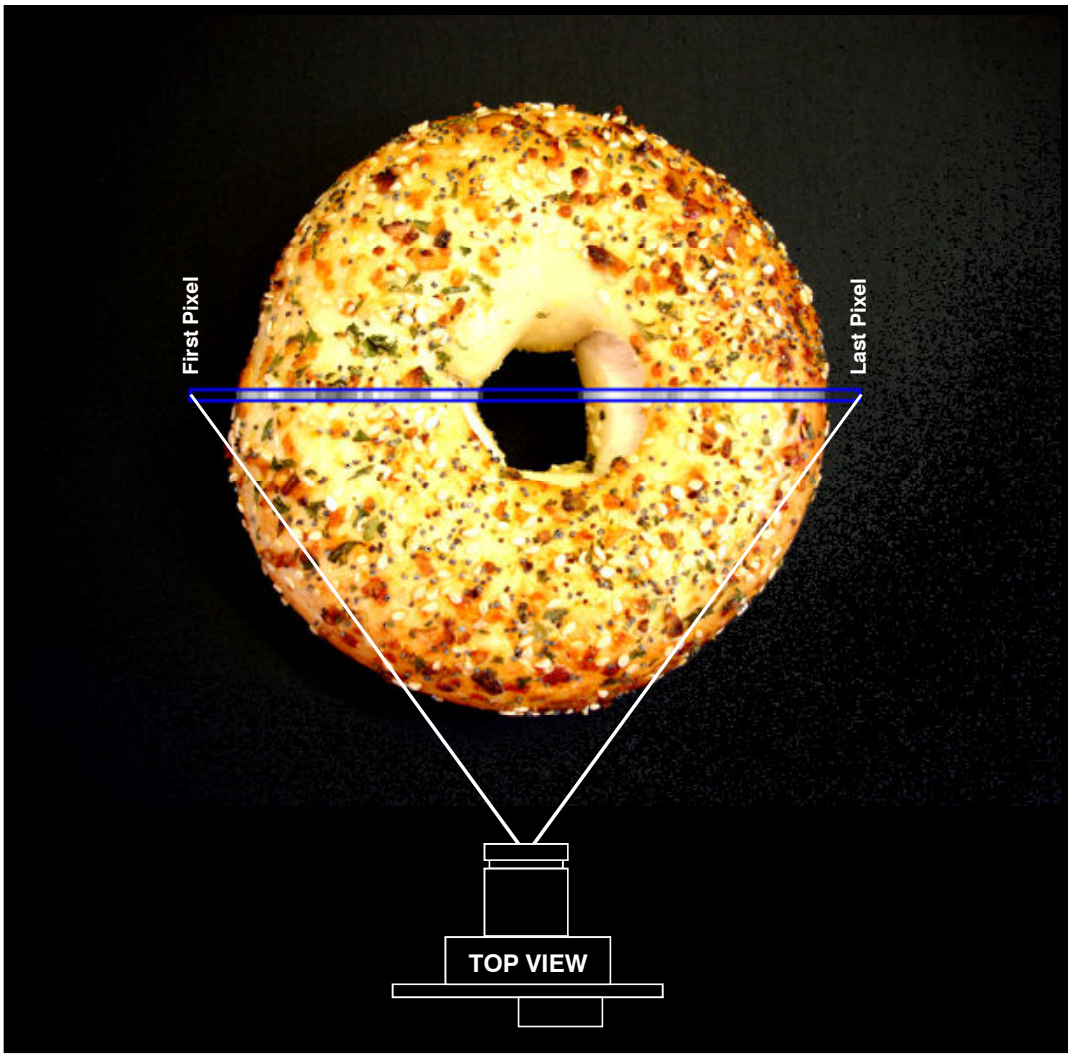
What You Need to Provide

- Parallax motherboard, or DB-Expander with BASIC Stamp and carrier board (such as the BOE).

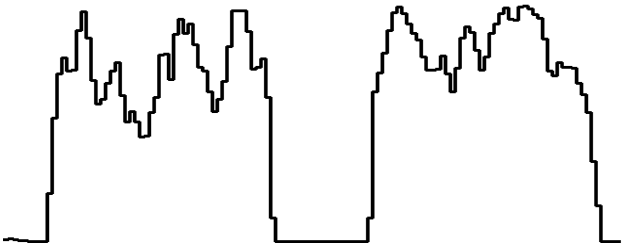
Introduction

What the Module Sees

The TSL1401R chip is a linear array (linescan) sensor. It consists of a single row of 128 photodetectors. The TSL1401-DB includes a lens to form images on the sensor array. What results is somewhat like peering through the narrow crack of a partially opened door to see a thin slice of what lies behind it. The illustration below helps to explain the concept:



The output from each observed pixel is an analog voltage proportional to light intensity. The analog intensity curve corresponding to the image above would look something like this:



Here, you can see not only the edges of the bagel and the hole in the middle, but also the intensity variations caused by the seeds and herbs on its surface.

The overall width (field of view) seen by the TSL1401-DB, using the included 7.9mm lens, is approximately equal to the subject distance. So, for example, if the module is 1 meter away from the subject, it will see a linear slice of the subject that's 1 meter wide and 1/128th of a meter high.

Focusing the TSL1401-DB's lens is accomplished by screwing it in or out. When screwed almost all the way in, distant subjects will be in focus. To focus on closer subjects, the lens needs to be screwed out a bit. Once proper focus is achieved, it may be necessary to secure the lens from vibration by wrapping tape around the lens bezel and lens holder barrel. If the lens is screwed in far enough, a small O-ring snapped into the crevice between the lens bezel and lens holder barrel will serve the same purpose.

Note: The use of a thread locker (e.g. Loc-Tite) or any cyanoacrylic adhesive (e.g. Super Glue) is *not* recommended near lens elements, as the fumes can destroy any optical coatings that may be present.

If you are using the TSL1401-DB with a Parallax MoBoStamp-pe, you can use the PC-hosted monitor program, described later in this document, as an aid to focusing.

Interface and Basic Operation

Refer to the schematic on the last page of this document for the TSL1401-DB's pinout, and to TAOS's TSL1401R-LF datasheet (available from www.taosinc.com) for the sensor chip's particulars. For normal operation (i.e. without external strobing or syncing), there are only three signals that need to be considered: **SI** (digital output to the sensor: begins a scan/exposure), **CLK** (digital output to the sensor: latches **SI** and clocks the pixels out), and **AO** (analog pixel input from the sensor: 0 – Vdd, or tri-stated if beyond pixel 128). The TSL1401 datasheet describes these signals in detail, so that description won't be repeated here, except as it relates to the BASIC Stamp.

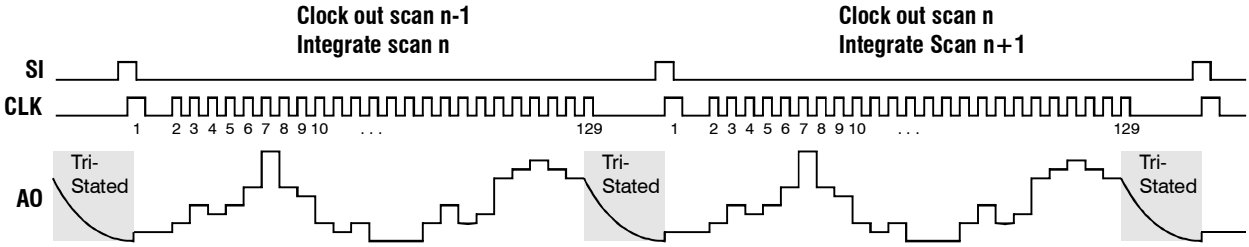
If you are using the TSL1401-DB with Parallax's DB-Expander (p/n 28325), the pin correspondences are as follows:

TSL1401R Pin	DB-Expander Pin
AO	A
SI	B
CLK	C

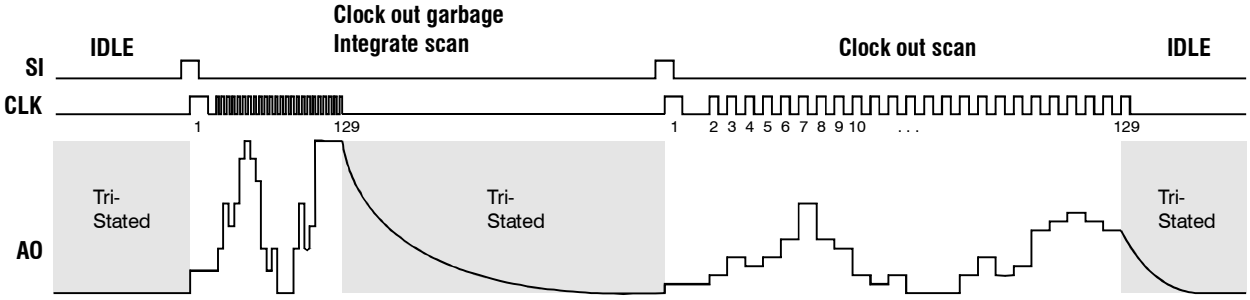
The TSL1401R is a *light-integrating* device. It's a bit like photographic film in that regard: the longer you expose it, the brighter the resulting image. Also, like film, it can saturate, such that if exposed too long, everything – even the darkest subjects – will look completely white. The exposure time (also called "integration time") is the time interval between **SI** pulses. (Well, actually, the exposure doesn't *really* begin until 18 clocks *after* **SI**; but it's often convenient to ignore that detail if those clocks occur quickly enough.) During each exposure, all the pixels need to be clocked out of the device to prepare it for the next exposure. However, the exposure interval for each pixel begins and ends with the **SI** pulse, not with the moment it's clocked out, as with some other sensors. Therefore, all the pixels get exposed simultaneously, and the acquired image represents the same interval in time for each of them.

There are two ways to acquire images with the TSL1401R: continuous and one-shot. In continuous imaging, the **SI** pulses occur in a steady stream, with 129 or more pixel clocks in between, during each exposure interval. To acquire an image, you need to wait for the next **SI** pulse time before clocking out

the pixels that constitute the image. These pixels will represent the light received during the previous exposure interval. Waveforms illustrating this method are shown below:



In one-shot imaging, the TSL1401R is left idle until it's time to snap a picture. Then **SI** clocked in, and 128 pixels are rapidly clocked out and discarded. Then you simply wait until the desired exposure time (since the **SI** pulse) has elapsed and pulse **SI** again. At this point, you can clock out the pixels resulting from the timed exposure. Here is a sample waveform:



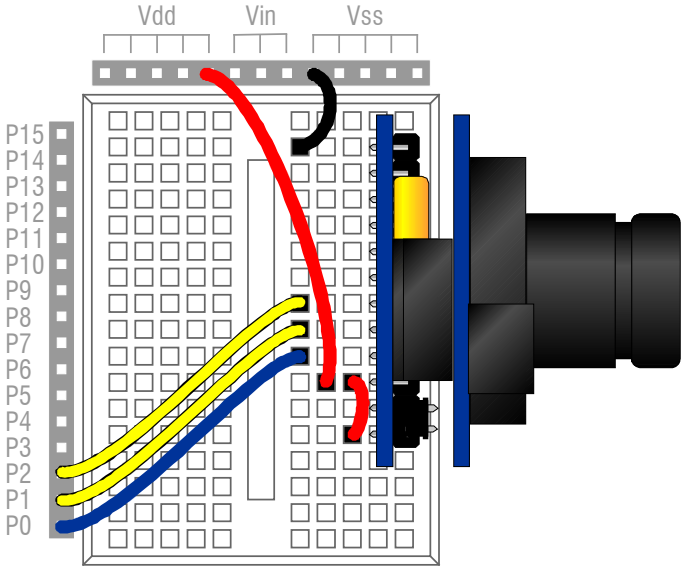
In all the discussion that follows, we will be using one-shot imaging.

Operation with the BASIC Stamp

This section explains how to use the TSL1401-DB directly with a BASIC Stamp. If you have the MoBoStamp-pe BASIC Stamp 2pe motherboard, you can skip this section and proceed to the section titled "Operation with the MoBoStamp-pe".

Connection

The following illustration shows how to connect the TSL1401-DB to a BASIC Stamp, using Parallax's Board of Education and a DB-Expander board:



The signal pinouts and port usage shown above are consistent with the examples to follow in this section. You can also use the DB-Extension Cable (p/n 500-28301) to separate the TSL1401-DB from the DB-Expander board if you need to.

Image Acquisition

The output of the TSL1401R is an analog signal, but the BASIC Stamp does not have analog input capability (except via RCTIME, which isn't fast enough to read 128 pixels). How, then, is it possible to use this device with a BASIC Stamp? We do it by connecting the **AO** signal directly to one of the Stamp's digital inputs. When done this way, the BASIC Stamp will threshold the analog input. Anything over about 2 volts will read as a **1**; anything under, as a **0**. By treating the signal this way, it's possible to input a string of **1**s and **0**s that represent light and dark portions of the "scene" being recorded. A complete scan, then, would require 128 bits of data (i.e. 16 bytes, or 8 words), which the BASIC Stamp can accommodate handily.

Once these bits have been read in, it's possible to analyze "features" of the scene by looking for groups of light and dark pixels. For example, if you wanted to measure the width of a light object against a dark background, you could read in the image, then count the number of "1" bits in the data. Likewise, if you wanted sense the edge of a "web" (e.g. paper in a paper mill) to keep the web on track, you would look for the first occurrence of a light or dark pixel in each scan.

The following PBASIC code fragment (taken from the complete program template shown later in this section) can be used to read a single scan from the TSL1401-DB. It consists of five lines of code, which are shown and discussed individually:

```
SHIFTOUT SI, CLK, 0, [1\1]
```

SI and **CLK** are defined in the larger program's preamble as **PIN**s and connect to like-named ports on the TSL1401-DB. This statement clocks out **SI** as a single bit of synchronous serial data, which starts a new exposure interval.

```
PWM CLK, 128, 1
```

We want to clock through all 128 pixels as fast as possible, since we're just timing an exposure here, not reading data. The **PWM** statement fits the bill perfectly. The **128** in this statement is the duty cycle (50%), not the number of pulses. The number of pulses is given by the **1**, which represents the length of time to output the PWM signal. For the BS2, this is nominally 1mS. (Actually it's more like 1.2mS and consists of about 150 cycles.)

```
PULSOUT SI, exp >> 1 MIN 1016 - 1016
```

This statement sets the exposure time. **exp** can be either a constant or a variable and represents the length of the exposure in microseconds. The reason for using **PULSOUT** instead of **PAUSE**, say, is that the timing resolution is so much finer. And the reason for sending the pulse on **SI** is that it's not used for anything else durng this time, and, so long as we don't clock the pulse with **CLK** the sensor chip is unaffected. The value subtracted from the pulse width (1016) represents the timing overhead from the PBASIC program, minus the start-of-integration delay to the 18th clock in the **PWM** statement. This means that the minimum exposure time will be about 2.032mS.

> **Note:** Timings for BASIC Stamps other than the BS2 will vary, and the value subtracted will need to be adjusted accordingly.

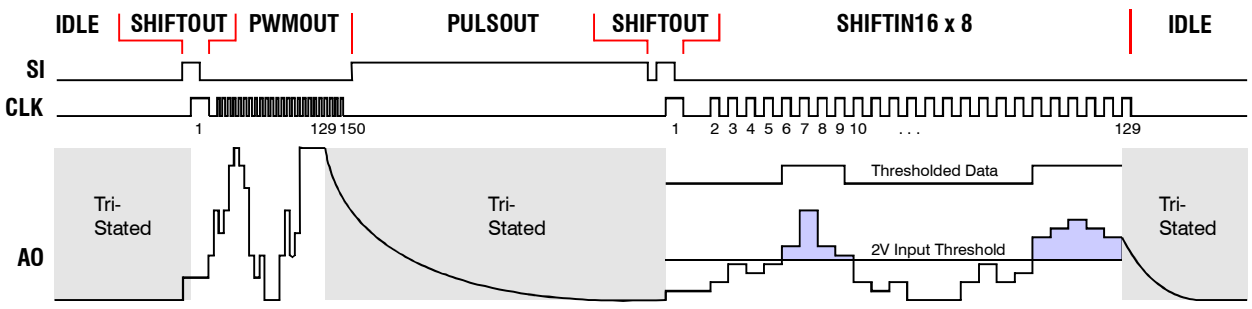
```
SHIFTOUT SI, CLK, 0, [1\1]
```

This clocks out the **SI** pulse again to end the exposure and begin actual data readout.

```
SHIFTIN AO, CLK, LSBPRE, [pdata(0)\16, pdata(1)\16, pdata(2)\16, pdata(3)\16]
SHIFTIN A0, CLK, LSBPRE, [pdata(4)\16, pdata(5)\16, pdata(6)\16, pdata(7)\16]
```

These two statements read 128 bits of thresholded pixel data from the **AO** pin into an eight-position word array, declared **pdata word(8)**, least-signficant bits first. Doing it "inline" like this is faster than doing it in a loop.

Here are the waveforms from the above acquisition routine, with the various sections annotated:



You can use **DEBUG** to display the acquired pixels, as in the following program fragment. (A somewhat fancier version is given in the complete program later in this section.)

```
FOR i = 0 TO 7
  DEBUG BIN16 pdata(i) REV 16
NEXT
```

The variable **i** can be declared as a **NIB**ble. The reason for the **REV** is because the data were read in LSB first, but **DEBUG**'s **BIN** formatter displays data **MSB**-first. So we need to reverse the order of the bits to get an accurate picture of the pixel order. One might well ask why we didn't just read the data in MSB first to begin with. After all, **SHIFTIN**, can do that just as easily. The answer lies in the image analysis routines to follow.

Image Analysis

Analyzing a linescan image to extract useful information from it involves two major operations: pixel counting, and pixel and edge location. The PBASIC subroutines that perform these operations treat the original array of eight words as an array of 128 bits, each bit corresponding to a single pixel. Bit 0 is the first pixel read; bit 127, the last. (This mapping is why the pixels needed to be read in LSB first.) Here's how the word and bit arrays are declared:

```
pdata   VAR Word(8)
pixels  VAR pdata.BIT0
```

Counting light or dark pixels within a given range is simple. Here's the code that does it:

```
CountPix:

  cnt = 0                                     'Initialize count.
  IF (lptr <= rptr AND rptr <= 127) THEN      'Valid range?
    FOR i = lptr TO rptr                      '  Yes: Loop over desired range.
      IF (pixels(i) = which) THEN cnt = cnt + 1 '     Add to count when pixel matches.
    NEXT
  ENDIF
  RETURN
```

cnt can be declared as a byte, since it will never exceed 128. **lptr** and **rptr** are also bytes that can range from 0 to 127, inclusive. They indicate the range over which the counting occurs. **which** is a bit variable that indicates whether to count dark pixels (**0**) or light pixels (**1**). Counting pixels is handy for computing an object's area – either in one scan, or cumulatively over multiple scans for two-dimensional objects passing under the camera on a conveyor.

Locating the first occurrence of a dark or light pixel within a given range isn't much harder:

```
FindPix:

  IF (found = 1 AND lptr <= rptr AND rptr <= 127) THEN
                                              'Still looking & within bounds?
    IF (dir = FWD) THEN                       '  Yes: Search left-to-right?
      FOR lptr = lptr TO rptr                 '         Yes: Loop forward.
        IF (pixels(lptr) = which) THEN RETURN '              Return on match.
      NEXT
    ELSE
      FOR rptr = rptr TO lptr                 '         No:  Loop backward.
        IF (pixels(rptr) = which) THEN RETURN '              Return on match.
      NEXT
    ENDIF
  ENDIF
```

```
  found = 0                                     'Didn't look or nothing found.
  RETURN                                        'Return.
```

found is a bit variable that should be initialized to **1** for the search to commence and indicates when the subroutine returns whether the desired pixel has been found (**0** = no; **1** = yes). **lptr** and **rptr** have the same meaning as in the counting routine, except that one or the other can get moved to the location of the found pixel. Combined with the cumulative effect that **found** has, this make it easier to perform a whole string of searches. **dir** is a bit variable that indicates which end of the (**lptr**, **rptr**) range to start the search from. You can predefine the constants **FWD** (= **0**, "left_to_right") and **BKWD** (= **1**, "right-to-left") to assign to **dir** to make your programs more readable. **which**, as with the counting routine, indicates what kind of pixel to look for. You can predefine constants for **which** as well (**DRK** = **0**; **BRT** = **1**) for readability.

When **FindPix** returns, **found** will indicate whether the desired pixel was located, and either **lptr** (if **dir = FWD**) or **rptr** (if **dir = BKWD**) will point to the found pixel location.

Sometimes, it's necessary to locate an edge instead of just a pixel. A *light edge*, for example is one that begins with a dark pixel, then transisitions to a light one. The **FindPix** routine can be used to find edges, too, by looking for the first pixel *opposite* of the edge you're seeking, then the next pixel after that that matches the edge value. The routine to do it is:

```
FindEdge:

  which = 1 - which  'Look for opposite kind of pixel first.
  GOSUB FindPix
  which = 1 - which  'THEN look for desired pixel.
  GOSUB FindPix
  RETURN
```

Locating pixels and edges is handy for finding objects in a field of view and measuring their "extents". An object's *extent* includes its outside boundaries and everything in between, regardless of pixel intensity. For example, in the bagel illustration, the bagel's extent would include the hole, while its *area* (obtained by counting bright pixels) would not.

Here is a complete program which incorporates all the routines described above (and then some). It acquires images and locates bright objects, computing both their extents and areas. You can also use it as a template for your own programs.

```
' =============================================================================
'
'   File...... TSL1401_scan.bs2
'   Purpose... Image capture and processing demo using the TSL1401-DB
'   Author.... Phil Pilgrim, Bueno Systems, Inc.
'   E-mail....
'   Started... 11 July 2007
'   Updated...
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =============================================================================


' -----[ Program Description ]-------------------------------------------

' This program demonstrates image capture and processing using the
' TSL1401-DB (Parallax p/n 28317). It continuously acquires and displays
' images from the TSL1401R sensor chip. It then locates both left and right
```

```
' bright edges, displaying them graphically using DEBUG. Finally it
' computes both the area and extent of the object found.

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

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

ao      PIN 0           'TSL1401R's analog output (threhsolded by Stamp).
si      PIN 1           'TSL1401R's SI pin.
clk     PIN 2           'TSL1401R's CLK pin.

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

DRK     CON 0           'Value assignd to "which" for dark pixels.
BRT     CON 1           'Value assigned to "which" for bright pixels.
FWD     CON 0           'Value assigned to "dir" for left-to-right search.
BKWD    CON 1           'Value assigned to "dir" for right-to-left search.

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

VariableName    VAR     Byte            ' What is variable for?

pdata   VAR Word(8)     'Pixel data, as acquired LSB first from sensor.
pixels  VAR pdata.BIT0  '128-bit pixel array mapped onto pdata.
exp     VAR Word        'Exposure (integration) time in 2uSec units.
lptr    VAR Byte        'Left pixel pointer for count and find operations.
rptr    VAR Byte        'Right pixel pointer for count and find operations.
i       VAR Byte        'General-purpose index.
cnt     VAR Byte        'Result of pixel-count routine.
which   VAR Bit         'Indicates seeking DRK or BRT pixels/edges.
dir     VAR Bit         'Indicates direction of search (FWD or BKWD).
found   VAR Bit         'Indicates pixels found (= 1), or not found (= 0).

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

' NOTE: This code assumes that DEBUG will wrap after 128 characters,
'       regardless of the DEBUG window width. Later versions of DEBUG
'       may not do this, and you will have to add CRs where needed.

exp = 8333              'Set exposure time to 8333uSec (1/120th sec).

DEBUG HOME              'Go to home position on DEBUG screen.
GOSUB DispHdr           'Display the pixel location header.

DO                      'Begin the scan-and-process loop.
  GOSUB GetPix          'Obtain a pixel scan.
  DEBUG CRSRXY, 0, 2    'Move to column 0, row 2.
  GOSUB DispPix         'Display the pixels here.
  lptr = 0 : rptr = 127: which = BRT : dir = FWD : found = 1
                        'Initialize parameters for find.
  GOSUB FindEdge        'Find first dark-to-light edge going L->R.
  dir = BKWD            'Switch directions.
  GOSUB FindEdge        'Find first dark-to-light edge going L<-R.
  DEBUG CLREOL          'Clear the next line.
  IF found THEN         'Both edges found?
    DEBUG CRSRX, (lptr - 1), "_|"    'Yes: Display left edge.
    DEBUG CRSRX, rptr, "|_"          '     Display right edge.
    GOSUB CountPix                   '     Compute area (light pixel count).
    DEBUG CR, CR, "Area = ", DEC cnt,'     Display area ...
      "   Extent = ", DEC rptr - lptr + 1, CLREOL '... and extent of object.
  ELSE                               'No:  Display failure message.
    DEBUG CR, CR, "No object found.", CLREOL
  ENDIF
```

```
LOOP

END

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

' -----[ GetPix ]---------------------------------------------------------

' Acquire 128 thresholded pixels from sensor chip.
' exp is the exposure time in microseconds.

GetPix:

  SHIFTOUT si, clk, 0, [1\1]           'Clock out the SI pulse.
  PWM clk, 128, 1                      'Rapidly send 150 or so CLKs.
  PULSOUT si, exp >> 1 MIN 1016 - 1016 'Wait for remaining integration time.
  SHIFTOUT si, clk, 0, [1\1]           'Clock out another SI pulse.
                                       'Read 8 words (128 bits) of data.
  SHIFTIN ao, clk, LSBPRE, [pdata(0)\16, pdata(1)\16, pdata(2)\16, pdata(3)\16]
  SHIFTIN ao, clk, LSBPRE, [pdata(4)\16, pdata(5)\16, pdata(6)\16, pdata(7)\16]
  RETURN

' -----[ DispHdr ]--------------------------------------------------------

' Display a header to aid in identifying pixel positions.

DispHdr:

  FOR i = 0 TO 12                    'Display tens digits.
    DEBUG DEC i DIG 0
    IF i < 12 THEN DEBUG "         " ELSE DEBUG CR
  NEXT
  FOR i = 0 TO 127                   'Display ones digits.
    DEBUG DEC i // 10
  NEXT
  RETURN

' -----[ DispPix ]--------------------------------------------------------

' Display 128 pixels: light pixels as "1"; dark, as "_".

DispPix:

  FOR i = 0 TO 127
    IF pixels(i) THEN DEBUG "1" ELSE DEBUG "."
  NEXT
  RETURN

' -----[ FindEdge ]-------------------------------------------------------

' Find the first edge within the range lptr through rptr, in the direction
' given by dir and of the type indicated by which (0 = light-to-dark;
' 1 = dark-to-light).

FindEdge:

  which = 1 - which                  'Look for opposite kind of pixel first.
  GOSUB FindPix
  which = 1 - which                  'Then look for desired pixel,
                                     '  by falling through to FindPix.

' -----[ FindPix ]-------------------------------------------------------
```

```
' Find the first pixel within the range lptr through rptr, in the direction
' given by dir and of the type indicated by which (0 = dark; 1 = light).

FindPix:

  IF (found = 1 AND lptr <= rptr AND rptr <= 127) THEN
                                          'Still looking & within bounds?
    IF (dir = FWD) THEN                   '  Yes: Search left-to-right?
      FOR lptr = lptr TO rptr             '         Yes: Loop forward.
        IF (pixels(lptr) = which) THEN RETURN '              Return on match.
      NEXT
    ELSE
      FOR rptr = rptr TO lptr             '         No:  Loop backward.
        IF (pixels(rptr) = which) THEN RETURN '              Return on match.
      NEXT
    ENDIF
  ENDIF
  found = 0                               'Didn't look or nothing found.
  RETURN                                  'Return.

' -----[CountPix]------------------------------------------------------------

' Count pixels within the range lptr through rptr, of the type indicated by
' which (0 = dark; 1 = light).

CountPix:

  cnt = 0                                 'Initialize count.
  IF (lptr <= rptr AND rptr <= 127) THEN  'Valid range?
    FOR i = lptr TO rptr                  '  Yes: Loop over desired range.
      IF (pixels(i) = which) THEN cnt = cnt + 1 '     Add to count when pixel matches.
    NEXT
  ENDIF
  RETURN
```

Pseudo-analog Pixel Acquisition

Even though pixels acquired by the BASIC Stamp are thresholded and converted to single bits, it's still possible to obtain a picture of each pixel's analog value – at least for static subjects that don't move. The voltage output from any given pixel in the TSL1401R can be expressed as follows:

Output voltage = k × LightIntensity × IntegrationTime

Where **k** is a constant. At the two-volt threshold level, this can be rewritten:

2 = k × LightIntensity × IntegrationTime

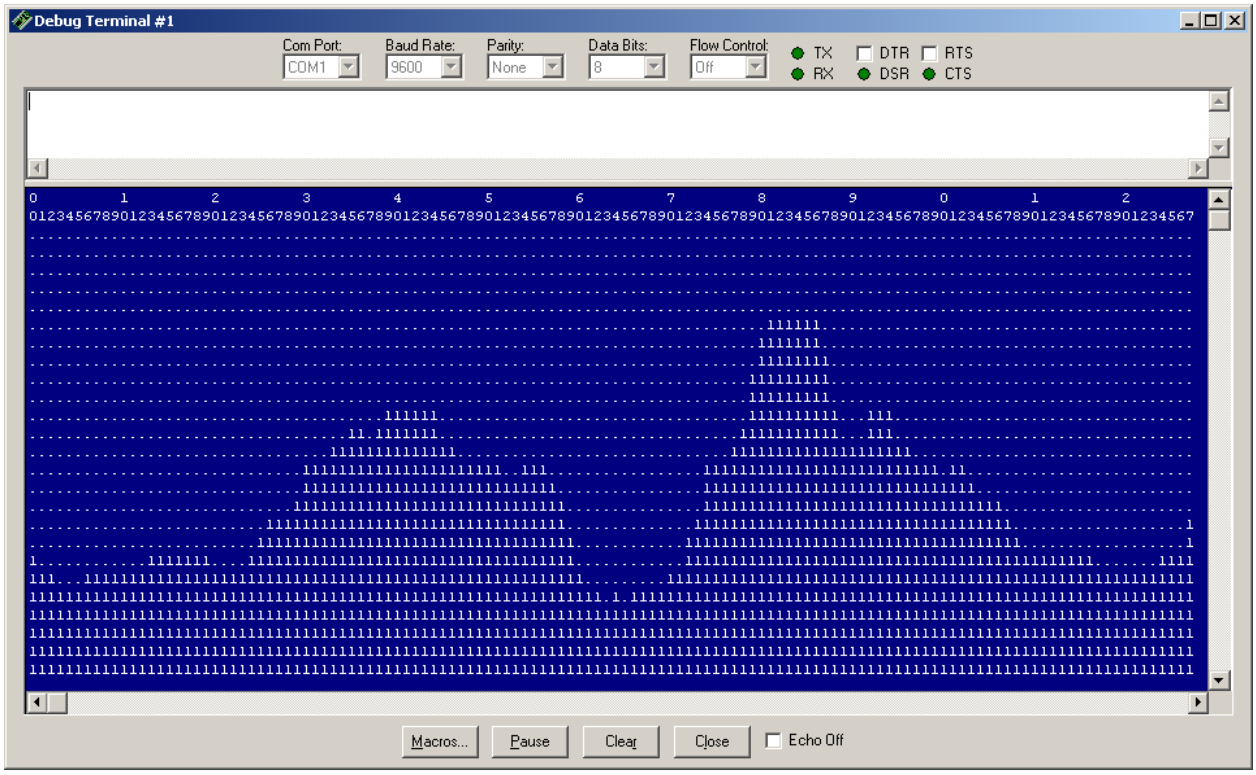
What we want this formula to answer is this: For a given light intensity to produce an output at the two-volt threshold, how long does the integration time have to be? Solving the above equation gives us our answer:

IntegrationTime = 2 / (k × LightIntensity)

This means that if we want all the pixels whose light intensity is at or above a certain level to read as ones, all we have to do is integrate for a time *inversely proportional* to that intensity level, and those pixels will read as ones after thresholding. By iterating over a range of intensity levels and displaying each line of pixels one above the other, we can obtain an analog bar graph of intensity levels. Here's the code snippet that does it. It can be plugged into the main code template shown above:

```
inten   VAR cnt

DEBUG HOME                'Go to home position on DEBUG screen.
GOSUB DispHdr             'Display the pixel location header.
FOR inten = 32 TO 8       'Cycle through intensities in reverse.
  exp = 32000 / inten * 8 'Compute exposure time as inverse of intensity.
  GOSUB GetPix            'Acquire the binary pixels.
  GOSUB DispPix           'Display them.
NEXT                      'Continue with next lower intensity level.
END
```

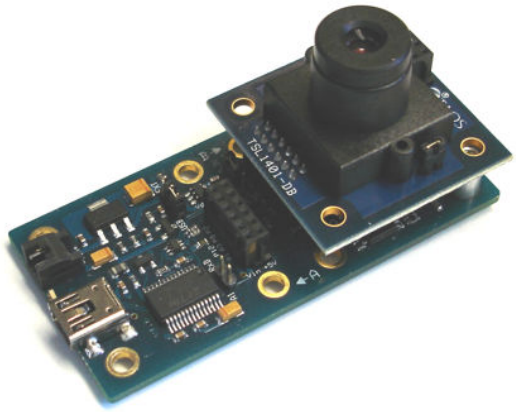
Here's what the output looks like when viewing a bagel lighted from the front:



You can see where the hole is, as well as the outside edges. However, this also illustrates a couple pitfalls of front lighting: specular reflection (glare) from the black background surface (i.e. black not being all that black), and rounded-off edges. Proper lighting techniques encompass an entire subject area of their own. The last chapter of this document links to various resources on the internet that cover this important topic.

Operation with the MoBoStamp-pe

The TSL1401-DB is designed to plug into the MoBoStamp-pe. Though it will work in either socket, it is recommended that socket "B" be used in order to make socket "A" available for interface-type daughterboards requiring access to Vin. In all the examples included here, socket "B" is assumed (and sometimes required). Here's a photo of the TSL1401-DB plugged into the MoBoStamp-pe, socket "B":



Loading the TSL1401R Driver Firmware

Before plugging in the TSL1401-DB, you will first want to load the firmware driver for it into the coprocessor associated with socket "B". Be sure you've downloaded the program "LoadAVR.exe" from the Parallax website, as well as the hex file for the TSL1401R driver, "TSL1401DB01.hex". Connect your MoBoStamp-pe to the PC's USB port, and then run LoadAVR.exe. Select "TSL1401DB01.hex" as the file to upload and socket "B" as the destination. Then click "Upload". Once the file has uploaded successfully, you can plug the TSL1401-DB into its socket.

TSL1401-DB Monitor Program

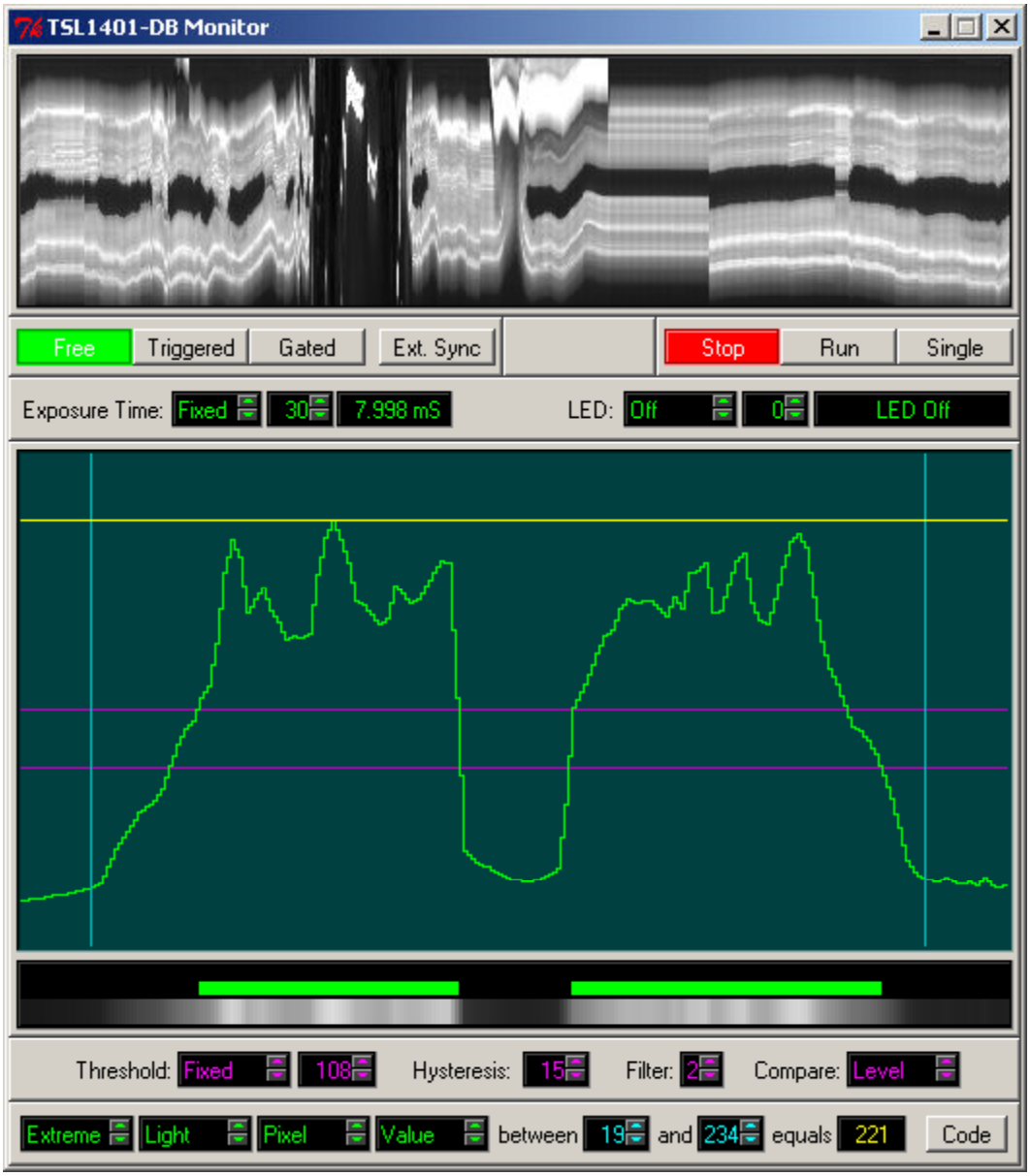
There is a Windows PC host program that will let you see what the TSL1401-DB sees in real time. It's called "TSL1401_monitor.exe", and it can be downloaded for free from the Parallax website. Just copy it to the directory of your choice.

With the MoBoStamp-pe/TSL1401-DB (the "camera") connected to your PC's USB port, start up the monitor program. After it makes a connection, it will upload its own PBASIC code, *which replaces any program currently in memory*. Then you should see something like what appears on the next page. Starting from the top, here are some points of interest in the display, some of which will be discussed in more detail later:

- **Scan Window:** Every scan from the camera gets appended to this image, from right to left, circulating back to the left edge when the screen is filled. By slowly rotating the camera on the motherboard's short axis, you can obtain a two-dimensional "scene". The bottom pixels in this window correspond to the leftmost pixels obtained in each scan.
- **Trigger/Sync Control:** Scans can be free-running, triggered, or gated. Triggering and gating are controlled by Pin 3, which is a BS2pe port common to both daughterboards. When triggering is set, a high-to-low edge on Pin 3 will cause a single scan to be acquired. When gating is set, scans will be obtained continuously while Pin 3 is low. Using the "Ext. Sync" button, scans can also be synchronized to an external source. The signal for this is obtained from the onboard 6-pin

mezzanine connector. The forthcoming LightSYNC-DBM plugs into this connector and provides a signal synchronized to the 50/60Hz variations in light level from fluorescent lamps. This makes the camera immune to these variations by starting all exposures at the same point on the 50/60Hz cycle. When syncing is used in conjunction with triggering or gating, the trigger/gate condition must be met first, *then* the sync pulse must be received.

- **Acquisition Control:** You can start and stop scanning with these controls or obtain scans one at a time.



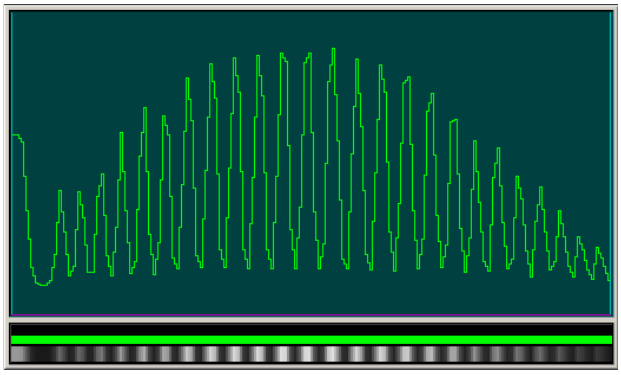
- **Exposure Controls:** These set the exposure type and time. Exposure times can be either fixed or automatic. When set to fixed, you can set the actual time using the numerical control. When set to automatic, exposure time is constantly adjusted to maintain the peak pixel value between certain bounds. In this case, the numerical boxes show the actual exposure time for any given exposure.
- **Lighting Controls:** When used with the forthcoming StrobeLED-DBM, which plugs into the 6-pin mezzanine socket, these controls adjust the type and timing of the light output from the LED.

Types are "Off", "Normal", and "Strobed". Normal mode means that the LED is on for the duration of each exposure at a brightness level that can be set in the numerical window. In strobed mode, the brightness level is fixed at maximum brightness, with the duration being the adjustable factor.

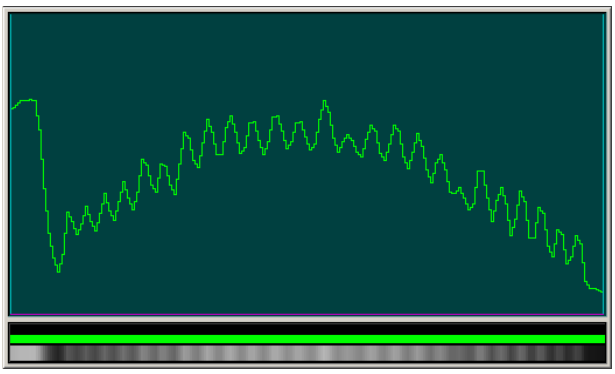
- **Scope Window:** This window shows the analog output of each pixel in real time. The horizontal magenta lines represent high and low thresholds for binary pixel acquisition, similar to the BS2 acquisition discussed in the previous section, but controllable. In normal comparator-style acquisition, the separation between these lines is the comparator hysteresis. These lines can be moved up and down by positioning the cursor between them and dragging them with the left mouse button held down. The separation (hysteresis) can also be adjusted, by dragging with the right mouse button held down. The vertical cyan-colored lines determine the area of interest for image analysis. They can be dragged left and right, individually, with the left mouse button held down. The yellow line – and any yellow feature, for that matter – represents the result of an image analysis measurement.
- **Image Brightness Window:** The narrow window below the scope window is divided into three slices. The bottom slice shows the instantaneous image brightness for each pixel as a gray level. The next slice up, shows which pixels register as "light" pixels after thresholding. The top slice is used to show image analysis results (in yellow) when those results include location or count information.
- **Binary Acquisition Controls:** These adjust the threshold types and values and are discussed in their own section below. Suffice it to say here that changes to these controls are reflected in the magenta lines displayed in the scope window.
- **Image Analysis Controls:** These controls make it easy to measure various features in an acquired image to test what might work or not work in your application. They are discussed in detail in their own section to follow. The value in the window after "equals" (in yellow) is the numerical result to which the yellow graphics coincide. In the example above, this is the value of the first pixel in the region defined by "between 1 and 255". The "Code" button will be used in a future rev of this program to write a PBASIC program for you that performs the scan acquisitions and performs the image analysis that you've selected.

Focus

The monitor program enables almost instant feedback for focusing the lens. Below are two screenshots: one of a backlit comb that's in focus, and one that's not. Notice the sharper edges and more pronounced detail in the in-focus image. This is what you need to strive for. By screwing the lens in and out, while watching the scope window, you will be able to maximize the sharpness of your image to obtain the best – and most repeatable – results.



Backlit Comb in Focus



Backlit Comb Out of Focus

Cosine Effect

You may have noticed in the images above that the center of the image is brighter than the edges. This is an optical property that's present in nearly all imaging systems. It's known as the "cosine effect", and it makes images appear brighter near their centers than at the edges. This happens because a light emitter, such as a diffuse backlight, is brighter on-axis than off-axis. Since the edges of a flat light source are captured more off-axis than the center, they appear darker. Compounding the effect is the fact that, behind the lens, light striking the sensor at the edges comes in at a more oblique angle than light striking the center. This effect becomes more pronounced as the imaging lens's focal length decreases (i.e. becomes more wide-angle).

Here's a scan of just the backlight, without anything in front of it. Even though the backlight itself is very evenly illuminated, it appears to have a cosine-shaped brightness contour when imaged with the camera.

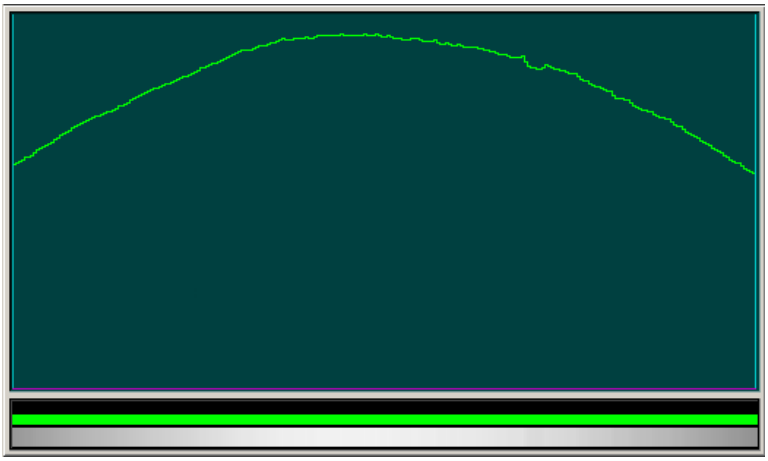


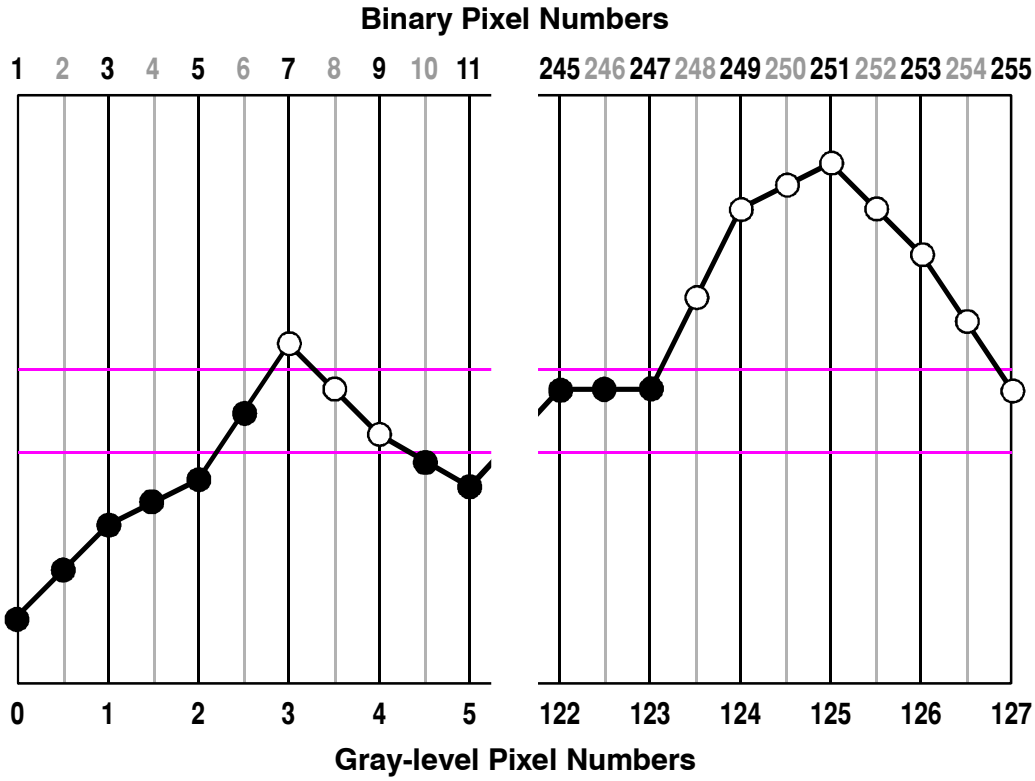
Image from Backlight Only, Showing the Cosine Effect

Different ways of dealing with this effect are discussed later in the section "Image Analysis and Measurment".

Note: This also illustrates the importance of keeping everything clean. Do you see that little divot in the trace, about two-thirds of the way across? It was caused by a tiny lint fiber clinging to the sensor chip. If you see something like this, unscrew the lens housing from the board, and use dry compressed air, or a soft cloth to remove whatever is causing the problem.

Binary Image Acquisition

The AVR firmware that you uploaded to the MoBoStamp-pe enables a wealth of image acquisition options, particularly in the conversion of grayscale pixel values to binary light/dark values suitable for image analysis. Binary pixels are acquired using *sub-pixel resolution*. This enables the acquisition of 255 binary pixels from the 128 grayscale pixels output from the TSL1401R. The firmware accomplishes sub-pixel resolution during image acquisition time by interpolating a virtual pixel between every pair of actual pixels, as shown below:

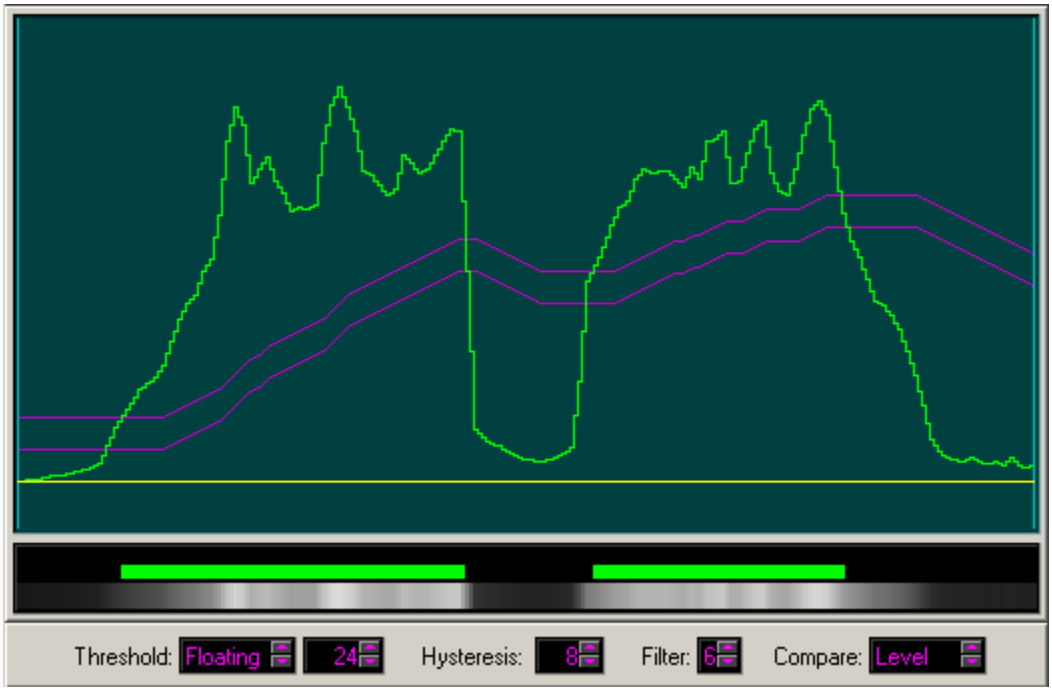


All light/dark thresholding takes place using the values of the 255 real and interpolated pixels, yielding 255 bits of data, whose positions are numbered **1** through **255**. (Note: This one-based numbering is different from that of the direct BS2 numbering shown earlier. That was zero-based to conform with PBASIC's zero-based subscript conventions.) Binary pixel **0** is non-existent in this system and is used in the context of image analysis to indicate "feature not found".

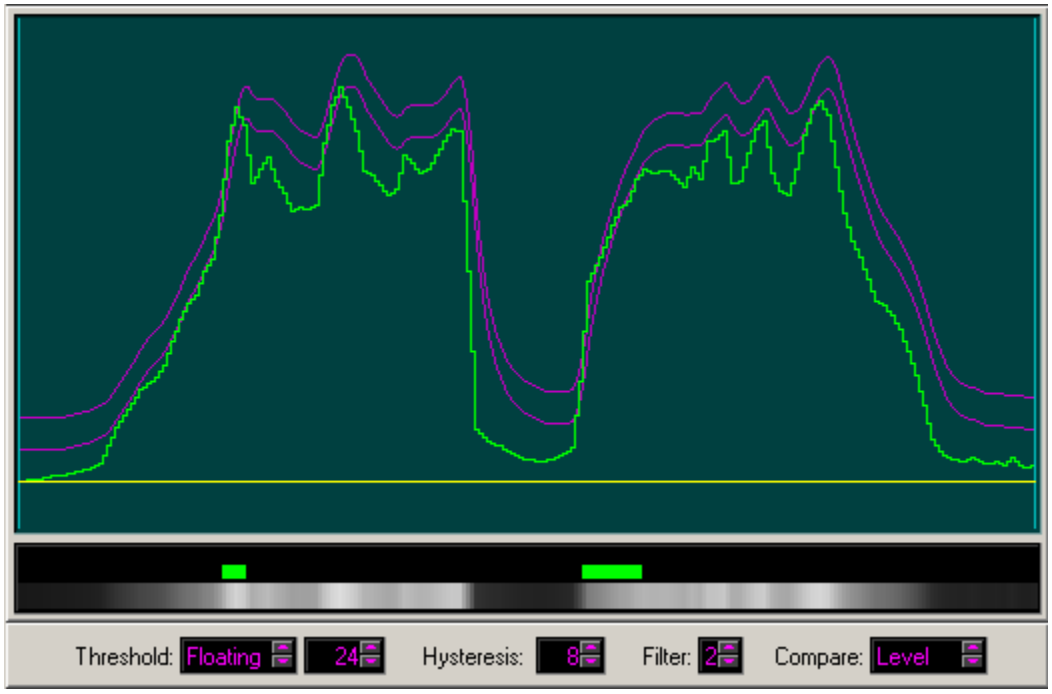
Conversion from gray-level to binary pixels always uses two thresholds for each pixel. Normal, compare-to-level thresholding treats the area between the two thresholds as a "hysteresis band". In order to transition from dark to light, a pixel must attain a light level above the upper threshold. To transition from light to dark, a pixel's level must sink below the lower threshold. Excursions into and out of the hysteresis band, without crossing it completely, will not result in a dark-to-light or light-to-dark transition. This helps to eliminate "hair trigger" transitions when the level is near threshold. (Of course, if this is what you want, you can always set the hysteresis value to zero.) This kind of comparison is illustrated in the above diagram by the color of the dots.

The firmware also supports "window" comparisons, in which values inside the hysteresis band evaluate to zero; those above or below, to one. This is useful for determining how much a subject's light intensity deviates from an acceptable range of levels, for example.

Thresholds can also be either "fixed", as the example above illustrates, or "floating". A floating threshold follows the contour of the pixel response as a kind of moving average whose filter constant is programmable. This can be handy for thresholding subjects whose illumination is uneven. It also allows the detection of extreme edges, while ignoring gently rising or falling light levels. The screen shots below illustrate this:



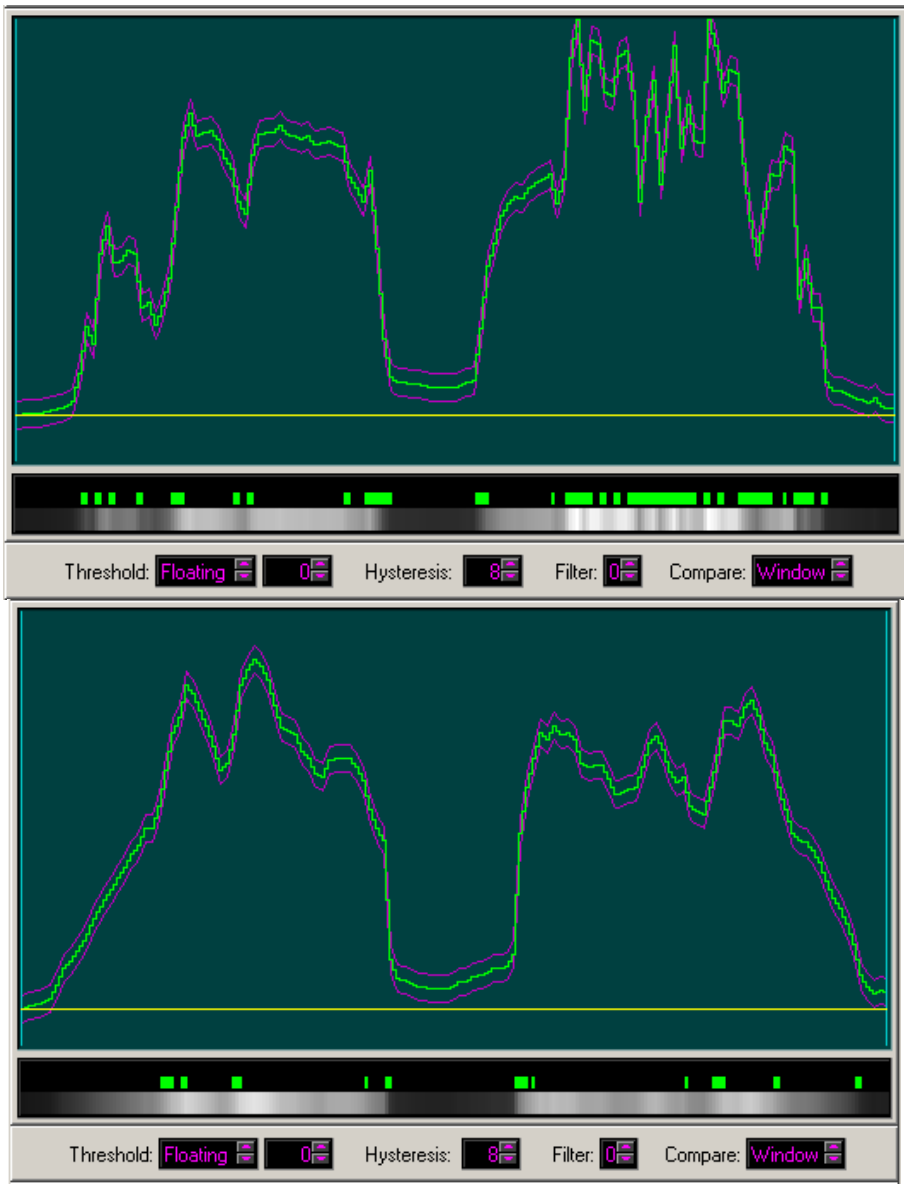
Here, a floating threshold with a filter factor of **6** has been selected. Now, see what happens when the filter factor is changed to **2**:



Only the most extreme rising edges are detected in this case, as indicated by the green area in the narrow strip below the scope window. In fact, one of the most important applications of floating thresholds is edge detection.

Another application of floating thresholds is in "texture" detection. Texture is a characteristic associated with rapidly alternating pixel values. The seeds and herbs on a bagel represent texture, for example. In

fact, a possible application might be inspecting bagels to see if they have enough "stuff" sprinkled on them. In this case, we would set the floating threshold level to zero, the filter to zero, and the hysteresis to a level consistent with how much texture we want to call "good". Also, we'll use window comparison, so that rising and falling pixel values get treated equally. Here is an example, using a bagel with stuff on it (top) compared to a plain bagel (bottom):



Notice how the green "one" pixels capture the texture of the coated bagel, and even the fact that the right side of that bagel has more stuff on it than the left side.

Image Analysis and Measurement

The monitor program is capable of performing feature measurement on binary images. A "feature" can be an edge location, the centroid of an object, the brightness of the brightest pixel, etc. The feature to be measured is selected on the measurement control bar at the bottom of the screen:

Possible values for the various options are:

Which	Type	Feature	Aspect	Between	And	Equals
First	Dark	Pixel	Value	1 to 255	1 to 255	Result
Last	Light	Edge	Location			
Extreme		Object	Count			
Average			Area			
Overall			Extent			

Not all combinations of these values will make sense or be realistic for the BASIC Stamp to compute. In such cases the result will be shown as "n/a". If a measurement *can* be computed, though, the numerical value will be shown in the "equals" box, and a graphical indicator (also in yellow) will be displayed at the appropriate place on the scope.

In a subsequent version of the program, the **Code** button will produce the PBASIC program necessary to acquire an image and make the desired measurement.

Now let's define some terms:

First: Beginning at the left-hand side, the first feature to match the conditions.
Last: Beginning at the left-hand side, the last feature to match the conditions.
Extreme Dark Pixel: Least bright pixel.
Extreme Light Pixel: Brightest pixel.
Average: Mean value of the feature(s) matching the conditions.
Dark Edge: Light-to-dark transition, reading from left to right.
Light Edge: Dark-to-light transition, reading from left to right.
Object: The span between an edge of the object type (dark/light) and an edge of the opposite type.
Value: The intensity of a pixel or collection of pixels.
Location: The pixel index (1 – 255) of the selected feature, or zero if the feature wasn't found.
Count: The number of features meeting the specified conditions.
Area: The number of dark or light pixels encompassed by the selected feature.
Extent: The number of total pixels encompassed by the selected feature.

When experimenting with the various measurement options, it's often handy to freeze image acquisition using the **Stop** or **Single** button. That way, you can adjust the measurement parameters with an image that's not itself changing.

Now, let's explore image analysis using a real-life example: bottling juice. Before the bottles are cased, the bottler wants to know two things: a) is the bottle full, and b) is the cap on? In this example, bottles will be passing between a backlight and the camera. So the camera will be looking through the bottle towards the backlight. This is what it will see. To the right of this image is a rotated scope display, showing what the TSL1401-DB sees. (Lighter is to the left; darker, to the right.)

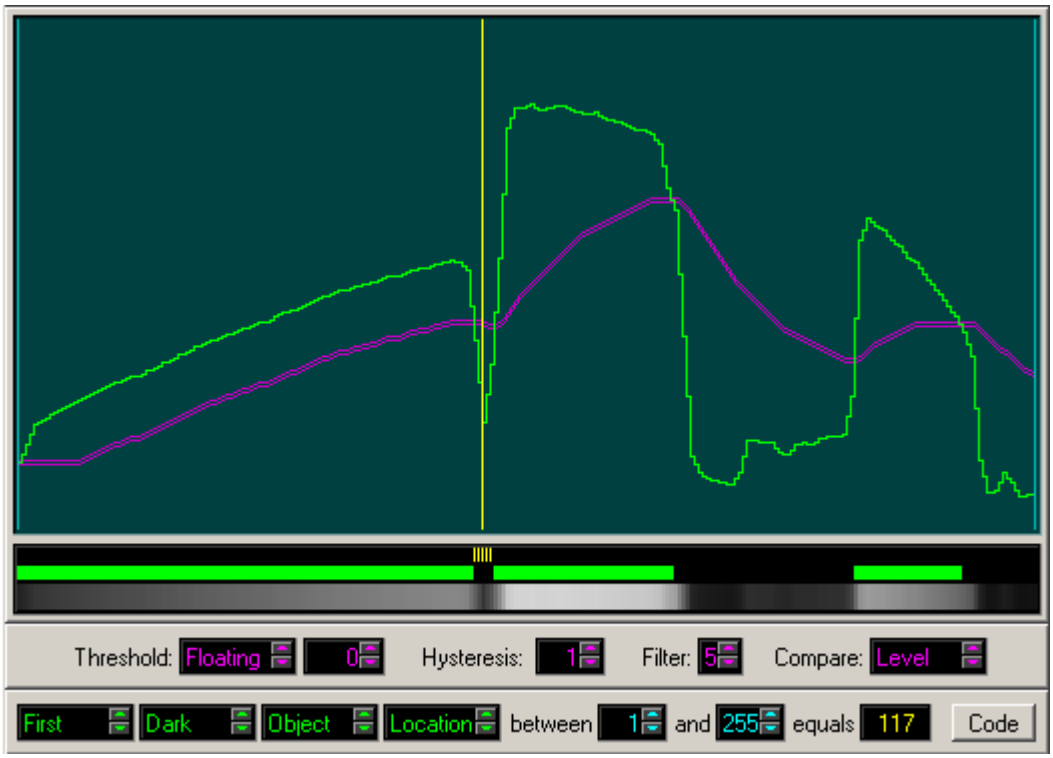
Clearly visible are the shadows created by both the cap and the meniscus at the liquid level. Although the juice in this example is colored, such a sharp meniscus will be present even with water-clear liquids. By measuring both the position of the meniscus and the size of the cap shadow, we can determine if the bottle is filled and capped properly.

Once again, you can see evidence of the "cosine effect" discussed earlier. There are multiple possible ways of correcting for this effect. These include:

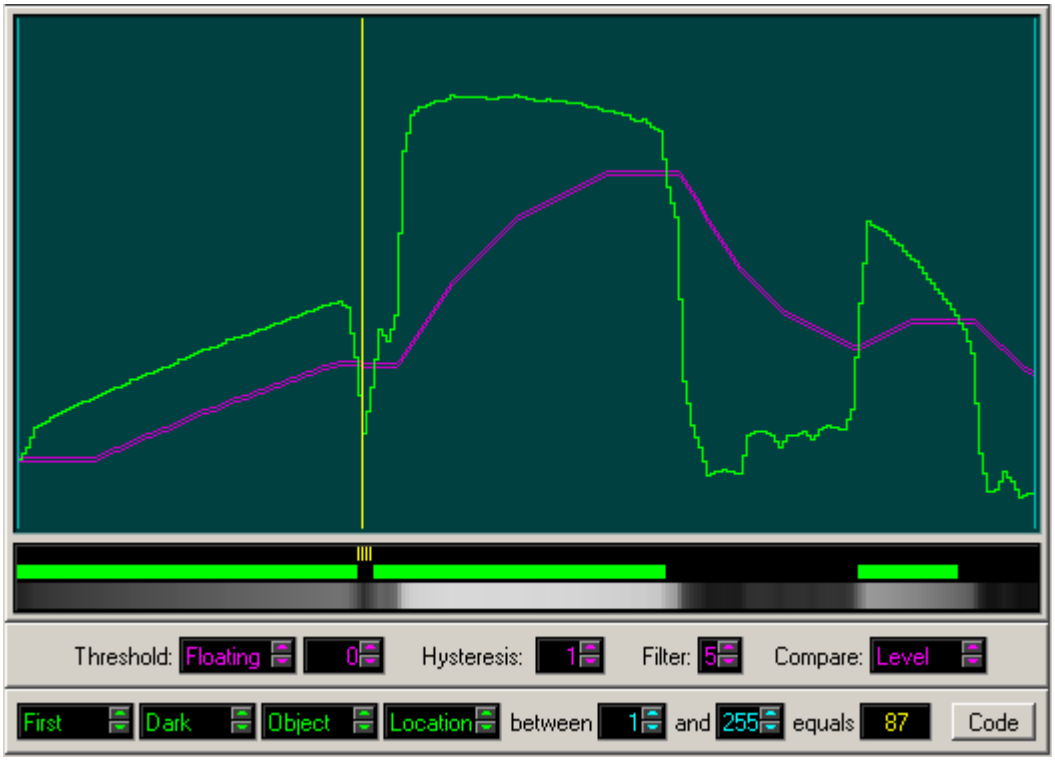
- Storing a brightness contour and dividing each captured pixel level by its contour value (not possible with the AVR firmware due to lack of memory).
- Using a telephoto lens and backing off from the subject to narrow the "capture angle" (possible with the TSL1401-DB, but requires a different lens).
- Using lighting that's brighter near the image edges than at the center (possible, but sometimes difficult).
- Using a floating threshold that approximates the contour when capturing binary images (easiest where practical).

In this example application, we will use the floating threshold method.

In the next pair of images, you can see the difference between a full bottle and one that's not so full. To find the liquid level we use the "First Dark Object Location" measurement:

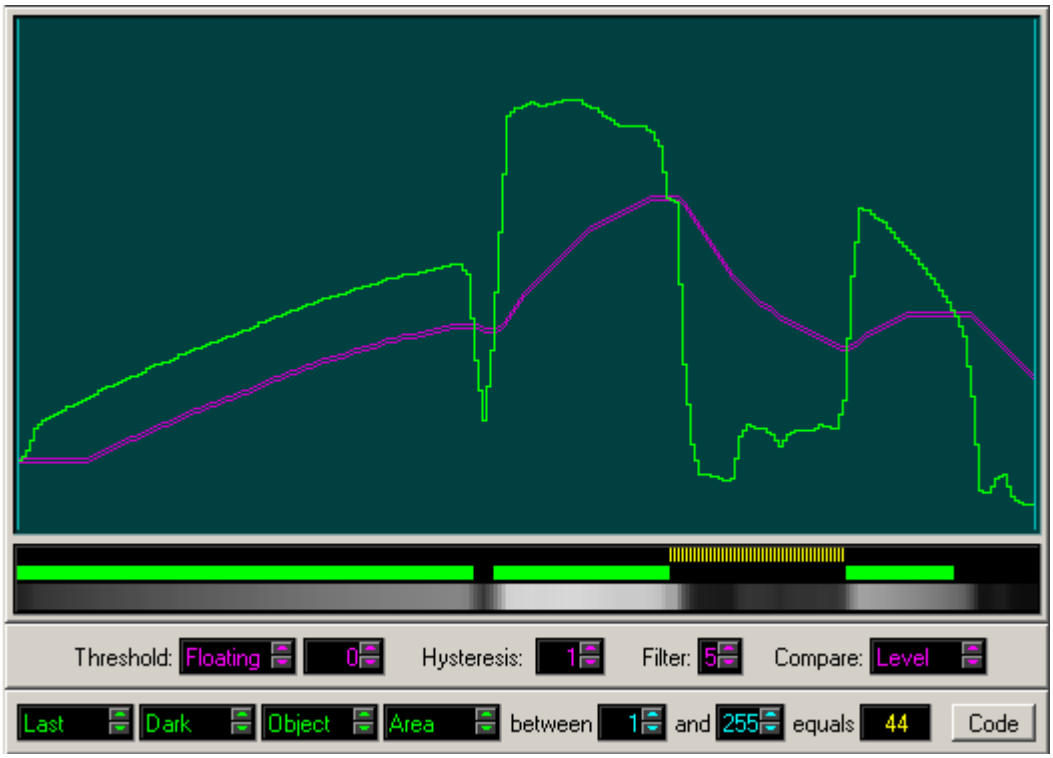


Full Bottle



Not-so-full Bottle

In the next pair of images, you can see the difference between a bottle that's capped and one that's not. Here we use the "Last Dark Object Area" measurement:



Capped Bottle

Uncapped Bottle

By setting allowable ranges for these two measurements, we can tell if any bottle presented to the camera "passes" inspection. In the next section, we will see how to write a PBASIC program to perform the various measurements required by applications similar to this one.

Programming with the TSL1401R Driver using PBASIC

Programs written in PBASIC for the MoBoStamp-pe can interact directly with the AVR coprocessor, which handles all the TSL1401R interface details. This interaction consists of sending commands to the AVR and waiting for results, which can then be read out to the user's PBASIC program for further action. There are commands for setting parameters, acquiring images, counting and finding pixels, and dumping results. Nearly all of the binary pixels processing can be done in the AVR itself at machine language speed, so it's seldom necessary to read the actual binary pixels into your PBASIC program. But they're available anyway if you need to examine them.

Immediate and Buffered Modes

Commands are handled by the AVR in two modes: immediate mode and buffered (deferred) mode. In immediate mode, you send the AVR a command and it is executed right away. In buffered mode, you can send as many as eleven bytes of commands, but they are not executed until the end-of-buffer command is received. This makes it possible to queue up commands to acquire and analyze an image ahead of time and then execute them all in rapid sequence when the proper moment arrives. Not all commands can be buffered, however. The ones that cannot are the ones that send data directly to the PBASIC program as they execute.

Sending Commands

Commands are sent to the AVR using PBASIC's **OWOUT** statement. For example, to begin acquisition of a simple binary image, you would write:

```
OWOUT owio, 0, [ACQBIN]
```

The pin designator, **owio**, is either **6** for socket B (preferred) or **10** for socket A. The designator, **ACQBIN**, is simply a constant defined in the code template at the end of this chapter. It's value is **$A4**. In all the examples that follow, we shall use these predefined constants, instead of their numerical equivalents, just to keep things as readable as possible. You will also want to use the code template (downloadable from Parallax's TSL1401-DB product page) to make writing – and reading – your programs easier.

Ready/Busy Polling

Some commands, such as the **ACQBIN** command mentioned above, require a finite amount of time to execute before their results can be read out or another command is sent. When these commands execute, the AVR needs to be polled until the "not busy" condition is detected. This applies only to immediate mode, though. In buffered mode, the "not busy" bit is sent only when all the commands in the buffer have finished executing. And it does this regardless of whether any of the commands would require it in immediate mode. Here's a snippet of PBASIC code that does the read/busy polling:

```
DO
  OWIN owio, 4, [busy]
LOOP WHILE busy
```

This reads a single bit variable, **busy**, and loops until it reads as a zero. It is most convenient to perform the busy checking in its own subroutine, since it may be required more than once in your program. The code template provides such a subroutine, named **Ready**.

Memory Map

Your PBASIC program has read-only access to 48 bytes of the TSL1401R driver's internal memory. This memory is used for storing binary pixel values, image acquisition stats, and the results of various image processing functions. It is laid out as follows:

Name	Addr	Description
PIXELS	**$00** to **$1F**	Binary pixel data. Pixels are packed LSB first. Since there are only 255 pixels, the last pixel (bit 7 of location **$1F**) is always **0**.
RESULTS	**$20** to **$24**	Beginning of the results buffer. The internal result pointer is set here after a reset and after state changes between immediate and buffered modes.
	$25 to **$2F**	Beginning of the command buffer (11 bytes) and continuation of the results buffer.

The named constants shown above are predefined in the code template near the end of this chapter. The 32-byte **PIXELS** area will always contain the results of the latest binary scan. Following that is the 16-byte **RESULTS** buffer. Immediately after an image is acquired, five bytes of the results buffer will contain statistics from the acquisition. These are discussed in detail in the image acquisition section. Unless the firmware gets an **OWIN/OWOUT** reset pulse or changes state, further results from image analysis are appended to these five bytes. The area they get appended to just happens to be the 11-byte command buffer, where buffered commands are stored. For this reason, the command buffer is only temporary storage, and a command sequence saved there must be reloaded each time it is used.

Resetting the Driver

PBASIC's **OWOUT** and **OWIN** commands include a provision for sending a "reset" pulse to the AVR. The TSL1401R driver will accept this pulse anytime it's expecting I/O from the BASIC Stamp (*and only then*) and will use it as a signal to reset its communication state and buffer pointers to their initial conditions. The reset pulse is most often used to terminate a sequence of result data being read from the AVR, as the following example shows:

```
OWOUT owio, 0, [DUMPADR, 32]                    'Begin dumping from buffer address 32.
OWIN owio, 2, [minpix, minloc, maxpix, maxloc] 'Read 4 byte variables.
```

Here **DUMPADR** is a command to begin dumping data from the address given by the next byte. It is followed immediately by a read, using **OWIN**. The **OWIN** parameter **2** indicates that a reset pulse should be sent *after* the statement is finished executing (i.e. after the four variables have been read out), which tells the AVR to quit transmitting data.

Any time such a reset pulse is received, the internal pointer that determines where the next result is deposited in memory is set to **$20**, the beginning of the **RESULTS** buffer. So, when you're reading results, make sure to read everything you need before sending a command that adds data to the buffer. Otherwise any remaining data that you need to read might be overwritten.

There are two instances when a reset pulse will not be recognized by the driver:

- When the driver is busy.
- When a triggered acquisition is awaiting the trigger pulse.

In the former case, just complete the not-busy polling before resetting the firmware. The latter case is discussed in the section, "Acquiring an Image".

Reading Data

Data can be read from the AVR using various forms of the dump command. The first is the **DUMPID** command, which reads the firmware ID and version number. Its format is:

DUMPID

DUMPID is a constant defined in the code template at the end of this section that has the value **$DD**. (In all code examples that follow, we shall use the defined constants, since they make the code so much more readable. It also makes the code more adaptable, in case the firmware gets upgraded and the hex commands change.) After sending it to the AVR using **OWOUT**, you can read three bytes of data: two of them are the letters "L" and "S" (for linescan); the last is just a byte (the version number), which, for this version, is **1**. Here's a snippet of code that reads and displays the firmware ID:

```
OWOUT owio, 0, [DUMPID]
OWIN owio, 0, [Ltr1, Ltr2, Ver]
DEBUG "Firmware version: ", Ltr1, Ltr2, DEC Ver
```

Ltr1, **Ltr2**, and **Ver** are Byte variables. When executed, the DEBUG screen should display:

```
Firmware version: LS1
```

The next dump command is **DUMPFLAGS** whose format is simply:

DUMPFLAGS

This command allows the PBASIC program to read one byte which contains various error flag bits that can be used for debugging. The format of the flag byte is as follows:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
BADCMD	**CANTBUF**	**CMDOVF**	**DATOVF**	**0**	**0**	**0**	**0**

The four most-significant bits are the error flags. A flag bit is set to **1** if its associated error has occurred since the last time **DUMPFLAGS** was executed. (When **DUMPFLAGS** is executed, all the error flags are cleared internally.)

- **BADCMD** is defined in the code template as **$80**. It can be ANDed with the flags byte to test whether the AVR has received an unrecognized command.
- **CANTBUF** is defined as **$40**. It can be ANDed with the flags byte to test whether an attempt was made to buffer a command (**ACQGRAY** or any of the dump commands) that can't be buffered.
- **CMDOVF** (command overflow) is defined as **$20**. When ANDed with the flags byte, it will tell you if an attempt was made to buffer more than 11 bytes of commands in the command/data buffer.
- **DATOVF** (data overflow) is defined as **$10**. When ANDed with the flags byte, it shows whether an attempt was made to buffer too many results in the command/data buffer.

Whenever an error condition occurs, the driver firmware will not permit further operations to be performed until a reset is received. Under these conditions, if you execute a **DUMPADR** before sending a reset, you will read a result equal to **$FF**. Since the last four bits of the result are supposed to be zero, you will know something is wrong and can then send a reset, followed by another **DUMPFLAGS** command to see what the error was. Here's the code that performs the aforementioned tasks:

```
OWOUT owio, 0, [DUMPFLAGS]
OWIN owio, 0, [flags]
IF (flags = $FF) THEN
  OWOUT owio, 1, [DUMPFLAGS]
  OWIN owio, 0, [flags]
ENDIF
```

The first **DUMPFLAGS** is sent without a prepended reset. Then the byte variable **flags** is read and compared with **$FF**. If it's equal, that means the firmware is waiting for a reset, so **DUMPFLAGS** is sent again – this time with a prepended reset (the **1** in the **OWOUT** command). At the end of this entire sequence, the variable **flags** will contain either the error flag, which can be ANDed with one or more of the error constants defined above to determine which error occurred, or zero, indicating that no error occurred. The above sequence is included in the code template as the subroutine **GetError**.

Finally, is the **DUMPADR** command, which is used to read results from the driver's memory. Its format is:

DUMPADR, Address

DUMPADR is a constant from the code template equal to **$DA**. Following it is an address byte, which can range from **0** to **47** (**$2F**). Once these two bytes are sent, the firmware expects your program to begin reading data using **OWIN**. It will continue sending data until a reset is received, at which point the internal address pointer is reset to **RESULTS** (**$20**). Here's an example for reading the average pixel value from the last scan:

```
OWOUT owio, 0, [DUMPADR, AVGPIX]
OWIN owio, 2, [average]
```

Here, **AVGPIX** is a constant from the template, equal to **$24**, and **average** is a byte variable used to hold the result.

Setting Exposure Time

The TSL1401R driver acquires all images using one-shot imaging, as described above in the "Interface and Basic Operation" section. It handles the exposure (integration) time details for you. All you have to do is tell it how long you want each exposure to be. This is done with the Set Exposure command:

SETEXP, ExpTime

SETEXP is a constant defined in the code template that follows, whose value is **$EE** (mnemonic for "enter exposure"). **SETEXP** requires one argument, **ExpTime**, the actual exposure time, which can range from **1** to **255** and represents a time span of 267µS to 68mS. **Note:** Because exposure timing is based on the AVR's internal RC clock, these times are approximate and can vary with temperature.

Here's a statement that sets the exposure time to **30** (about 8mS):

```
OWOUT owio, 0, [SETEXP, 30]
```

Once the exposure time is set, it stays set until changed by another **SETEXP**. If you never set the exposure time explicitly, it defaults to a value of **128**.

Setting Binary Acquisition Coefficients

When a binary image is acquired, each pixel is first read from the AVR's A/D converter as a value between 0 and 255. Then it's converted to a **0** or a **1**, depending on the values of the three coefficients

provided by this command. These are the **Threshold**, the **Hysteresis**, and the **Mode**. These coefficients are the same as described above in the "TSL1401-DB Monitor Program" section. The sequence of bytes required by the Set Binary Coefficients command is:

SETBIN, Threshold, Hysteresis, Mode

Where **SETBIN** is a constant having the value **$EC** (mnemonic "enter coefficients").

Threshold is the value which determines whether an acquired pixel is a one (light) or a zero (dark). **Hysteresis** is the width of the band above and below **Threshold**, which, in effect, creates two thresholds: an upper and a lower. A pixel must be at least as high as the upper threshold to cause a transition from dark to light; and it must be lower than the lower threshold to cause a transition from light to dark. The separation between the two thresholds (the hysteresis band) is twice the value entered for **Hysteresis**.

The **Mode** byte includes flags that determine the binary acquisition mode, along with the floating threshold **Filter** value. It's format is:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0: Fixed 1: Floating	0: Level 1: Window	Reserved	Reserved	Reserved	Filter value for floating threshold: 0 (no filtering) to 7 (maximum)		

When **FIXED** thresholding is selected, each pixel is compared to the same, constant, upper and lower thresholds, defined by **Threshold** and **Hysteresis**. The first pixel is compared to **Threshold** alone to determine the initial state (**0** or **1**). After that, each pixel is compared with either **Threshold + Hysteresis**, if the last comparison yielded a **0**, or **Threshold – Hysteresis**, if the last comparison yeilded a **1**. Both **Threshold** and **Hysteresis** can range from 0 to 255. When their sum is greater than 255, 255 is used as the upper threshold. When their difference is less than 0, 0 is used as the lower threshold.

When **FLOAT**ing thresholding is selected, the value given for **Threshold** is assumed to be a signed byte that ranges from **–128** ($80) to **127** ($7F). It is treated as an offset, which is added to an internal floating parameter (**Float**) that changes, depending on the values of the pixels that preceded it. For the first pixel, **Float** is simply assigned the value of that pixel. From there on out, **Float** is modified after each pixel is processed, according to the formula:

Float = Float + (Pixel - Float) / (1 >> Filter)

Therefore, when **Filter** equals **0**, **Float** takes on the value of the pixel itself. When **Filter** is a larger number, **Float** becomes a moving average whose time constant increases as **Filter** increases. The new value of **Float** is used to process the *next* pixel.

Each pixel is then compared with **Float + Threshold + Hysteresis**, if the previous pixel evaluated to a **0**, or with **Float + Threshold – Hysteresis**, if the previous pixel evaluated to a **1**.

The preceding discussion assumes that **Level** thresholding is in effect. If **Window** thresholding is selected instead, the upper and lower thresholds are computed as described above, but each pixel is assigned a **0** if it's value lies between the two thresholds, and a **1** if it's value lies above the upper threshold or below the lower one (i.e. outside the window formed by the hysteresis band).

Here is a chart that shows how each binary acquisition mode might be used:

Threshold	Comparison	Filter	Application
Fixed	Level		Looking for objects and edges when lighting is even.
Fixed	Window		Looking for intensity excursions outside a certain band.
Floating	Level	5-7	Looking for objects and edges when lighting is uneven.
Floating	Level	2-4	Edge location among widely varying pixel values
Floating	Window	0-1	Looking for texture, whose minimum intensity is determined by **Hysteresis**. Set **Threshold** to **0**.

Here's a statement that sets **Threshold** to 30, **Hysteresis** to 10, and uses a floating threshold with level comparison and a filter value of 5:

```
OWOUT owio, 0, [SETBIN, 30, 10, FLOAT|LEVEL|5]
```

The constants **SETBIN**, **FLOAT**, and **LEVEL** are defined for you in the code template near the end of this section. Instead of **FLOAT**, you could also choose **FIXED**; and instead of **LEVEL**, **WINDOW**, which are also predefined.

Setting the LED

If you are using the StrobeLED-DBM mezzanine board (forthcoming), you can set the duration and/or brightness of the LED flashes as shown here. The basic format for this command is:

SETLED, Amount

Where **SETLED** is a constant equal to **$EB** (mnemonic "enter brightness"), and **Amount** has the following format:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0: Intensity	LED brightness value (0 – 127), equivalent to 0 – 50% for duration of exposure.						
1: Time	LED "on" (strobe) time (0 – 127), equivalent to 0 – 3.4mS at 100% brightness.						

Once the **SETLED** command has been issued, it stays in effect until reissued. When the lower seven bits of **Amount** are zero, the LED is effectively turned off. Otherwise, it is turned on at the beginning of each exposure. If bit 7 of **Amount** is zero, it remains on for the entire integration time at a level selected by the lower seven bits. If bit 7 is one, it strobes on at 100% brightness for the duration specified by the lower seven bits. This time is approximate and is governed by the AVR's internal RC clock. So it will vary somewhat with temperature.

Here's some sample code that causes the LED to strobe for about 1mS at the beginning of each exposure:

```
OWOUT owio, 0, [SETLED, TIME|75]
```

TIME is a constant defined in the code template that equals **$80**. In its place, the constant **INTEN** may be used. It's equal to zero and does nothing, but it makes the code more readable.

Acquiring an Image

Image acquisition is done using the various acquire commands, each having the same general format:

Acquire
Acquire|XTRIG

They all take no arguments and come in eight flavors, which are described in the table below:

Command	Value	Description
ACQGRAY	**$A0**	Acquire a binary image, while dumping all 128 grayscale pixel values.
ACQBIN	**$A4**	Acquire a binary image, replacing the previously-acquired image.
ACQAND	**$A1**	Acquire a binary image, ANDing the binary pixels with the previous image. This can be used to track which bright pixels are common to all successive scans.
ACQOR	**$A2**	Acquire a binary image, ORing the binary pixels with the previous image. This can be used to track which bright pixels appear in one or more successive scans.
ACQXOR	**$A3**	Acquire a binary image, XORing the binary pixels with the previous image. This can be used in successive pairs to see which pixels change between them (useful for motion detection).
ACQORNOT	**$A5**	Acquire a binary image, ORing the binary pixels with the NOT of the previous image. This reveals only those pixels that *become* dark betweens pairs of scans.
ACQANDNOT	**$A6**	Acquire a binary image, ANDing the binary pixels with the NOT of the previous image. This reveals only those pixels that *become* bright between pairs of scans.
ACQXORNOT	**$A7**	Acquire a binary image, XORing the binary pixels with the NOT of the previous image. This can be used in successive pairs to see which pixels remain the same between them.

Two additional constants are defined in the code template: **ACQDIFF**, which is a pseudonym for **ACQXOR**, and **ACQSAME**, which is a pseudonym for **ACQXORNOT**. These pseudonyms reflect the use of these two commands, which is to find pixels which are either different from, or the same as, pixels from the previous acquisition.

Although the acquire commands do not take an argument, there is one optional modifier that can be ORed to it. This is the external trigger constant, **XTRIG**, which has a value of **$08**. When **XTRIG** is ORed to any acquire command, the command, when executed, will wait for a falling edge on BASIC Stamp pin **P3**, then begin its exposure. Since **P3** is common to both daughterboard sockets, this same pin triggers the TSL1401-DB in either one. This enables exposures to be synchronized precisely with an external event, such as an encoder pulse or optosensor output.

Note: When the driver firmware is waiting for an external pulse on **P3**, it is not possible to reset it using the **OWOUT** statement's reset pulse. However, you can force an exposure by pulling **P3** low momentarily with the following PBASIC code sequence:

```
LOW 0 : INPUT 0
```

Note that **P3**, like **P2**, should not be driven high. The MoBoStamp-pe includes pull-ups for this purpose, allowing bussed open-collector drivers to actuate it by pulling it down.

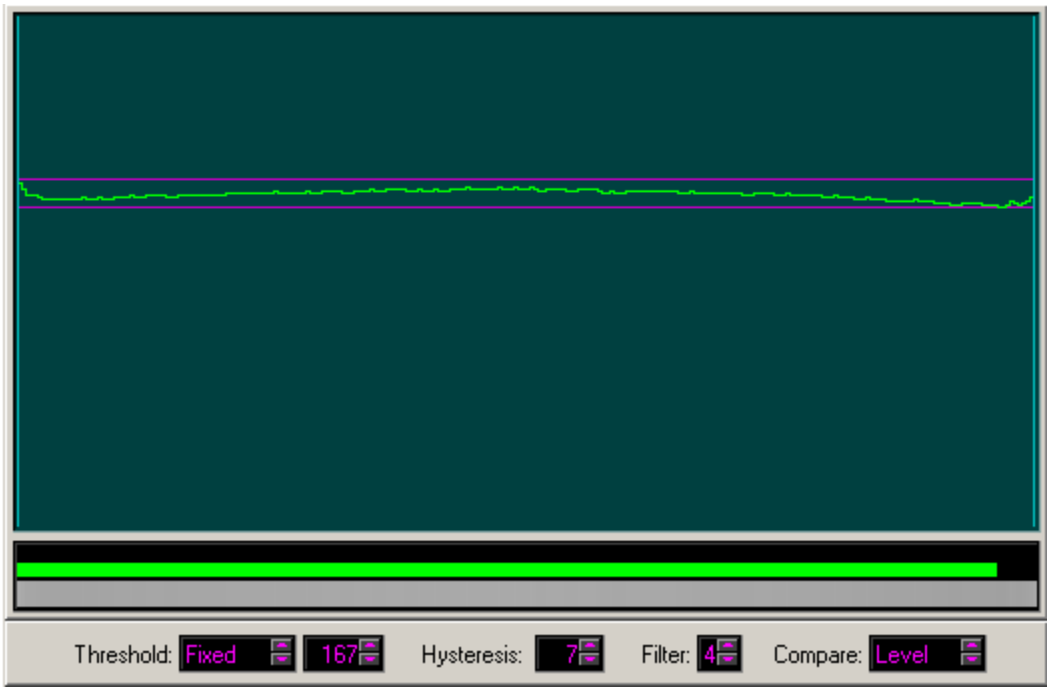
All of the acquire commands send out a not-busy bit in immediate execution mode when they have completed their work. The exception is the **ACQGRAY** command, which sends the not-busy bit ahead of outputting the 128 bytes of grayscale pixel data. In any case, it is necessary to poll for this bit after sending any acquire command to the AVR.

The image acquisition commands all result in 255 bits of subpixel-resolution image data in the AVR's internal data buffer. You can access this data using the **DUMPADR** command described later. However, this is seldom necessary, given the firmware's internal image analysis functions, also described later. Here's a code snippet that sets the exposure time to 30, the threshold and hysteresis to a fixed 64 and 10 respectively, causes the AVR to wait for **P3** to transition low, then acquires a binary image. It then waits for the image acquisition to complete by calling the **Ready** subroutine (defined in the code template) that polls the AVR for a not-busy condition:

```
OWOUT owio, 0, [SETEXP, 30, SETBIN, 64, 10, FIXED|LEVEL, ACQBIN|XTRIG]
GOSUB Ready
```

This also demonstrates how commands can be chained in a single **OWOUT** statement.

The **ACQGRAY** command is unique in that it gives you access to the gray-level pixel values as it reads them out from the TSL1401R. Since there isn't enough memory in the AVR to store all these values, they have to be sent to the BASIC Stamp on the fly. For this reason, any PBASIC program that uses **ACQGRAY** should read these 128 bytes of data and do something with them as quickly as possible. Slowing down acquisition from the TSL1401R chip could result in "pixel droop", as the internal charge storage capacitors self-discharge. The two plots below illustrate this effect. For this test, the lens and lens housing were removed in order to illuminate the TSL1401R chip as evenly as possible. In the first, the pixels are read out at and transmitted to the host PC at maximum speed (about 1.4 mS/pixel). The pixel intensities are all within 14 of each other (8% of the maximum), as indicated by the threshold cursors. In the second, an additional 20mS was added to each pixel time (2.56 seconds overall) to read out the scan. You can see the droop near the end that results from the internal caps discharging – very unevenly – as they await their turn to be read. And the band between the highest and lowest pixels increases to 48, or 27% of the maximum. Of course, two and a half seconds to read out the pixels would be quite extreme. But it illustrates quite graphically what happens if they're read out to slowly. Also, bear in mind that this effect can only occur with **ACQGRAY** and not with any of the binary acquisition commands.



Intensity plot resulting from even illumination and a fast readout. The little "blips" at the ends are due to a lensing effect from the edge of the clear chip package. They do not appear when an imaging lens is in place.

Intensity plot resulting from even illumination and a very slow readout.

Here's a program snippet that uses **ACQGRAY** and outputs the pixel data at 38400 baud to the DEBUG port. It's identical to that used by the TSL1401-DB Monitor Program:

```
pixno VAR Byte
char  VAR Byte(16)

OWOUT owio, 0, [ACQGRAY]
GOSUB Ready
FOR pixno = 0 TO 7
  OWIN owio, 8, [STR char\16]
  SEROUT 16, 6, [STR char\16]
NEXT
```

In the above code, **Ready** is a subroutine, defined in the code template, which polls the AVR until it's no longer busy.

The binary acquisition commands that perform Boolean operations on the data acquired from a previous scan can be used to see what changes or stays the same between acquisitions. With the possible exception of **ACQAND** and **ACQOR**, they are typically used second in a pair of commands, the first being an **ACQBIN**. In other words, you first obtain a scan that simply records pixels in the usual way, then wait awhile, then obtain another scan modifies the first one.

In addition to the 32 bytes of binary pixel data, each acquire command also collects statistics from the image it has acquired. These are the intensity and location of the dimmest pixel, the intensity and location of the brightest pixel, and the average pixel intensity over the entire image. The following chart shows the readable AVR memory locations, including those affected by the acquisition commands:

Name	Addr	Description
PIXELS	**$00** to **$1F**	Binary subpixel data. Pixels are packed LSB first. Since there are only 255 pixels, the last pixel (MSB of location **$1F**) is always **0**.
MINPIX	**$20**	Intensity (0 – 255) of the dimmest pixel.
MINLOC	**$21**	Location (0 – 127) of the dimmest pixel. If multiple pixels share the lowest intensity, it will be the location of the last one.
MAXPIX	**$22**	Intensity (0 – 255) of the brightest pixel.
MAXLOC	**$23**	Location (0 – 127) of the brightest pixel. If multiple pixels share the brightest intensity, it will be the location of the last one.
AVGPIX	**$24**	Average pixel intensity (0 – 255) of all 128 pixels.

The location information ranges from **0** to **127**. This is because we're dealing with grayscale pixels here, and there are only 128 of them, whose locations are zero-based. (Binary pixels, by contrast, number from **1** to **255**.) The information provided by these stats can be used, for example, to adjust the exposure time to maintain a constant maximum pixel intensity. An example of this technique is shown in a later section.

In addition to buffering pixels and stats, the acquire commands initialize the internal Left and Right pointers for counting a searching to **1** and **255**, respectively.

Counting Pixels and Edges

One of the simplest methods of analyzing a binary image is to determine how many pixels are bright and how many are dark. This information alone can tell you an object's width, for example, a conveyor's degree of coverage, or whether a bottle cap is present. The commands that performs this task are **CNTNEW** and **CNTNXT**:

CNTNEW|Modifiers, Begin, End
CNTNXT|Modifiers

CNTNEW is a constant defined in the template as **$C8.** The **CNTNEW** command counts pixels or edges between pixels **Begin** and **End**, inclusive. **Begin** and **End** can range from **1** to **255**, and **Begin** should be less than or equal to **End**. (Remember that binary pixel locations are one-based, not zero-based. The first pixel is pixel #1.)

CNTNXT (**$C0**) counts pixels or edges between the current internal Left and Right limits, whatever they might be. These limits are set by the following actions, whichever occurred most recently:

- Any acquire command, which resets Left to **1** and Right to **255**.
- Any **CNTNEW** command. Left ends up at **Begin**; Right, at **End.**
- A **FNDNEW** command. Left is initialized to **Begin**; Right, to **End**.
- Any **FNDNEW** or **FNDNXT** command, wherein Left is moved to the found pixel or edge (searching forward), or Right is moved to the found pixel or edge (searching backward). If the desired feature is not found, Left will equal Right + 1, causing further counts and finds to return zero.

The modifiers that can be used with (i.e. ORed to) **CNTNEW** or **CNTNXT** are:

Name	Value	Description
DRKPIX	**$00**	Count dark pixels.
BRTPIX	**$02**	Count bright pixels.
DRKEDG	**$03**	Count dark (high-to-low) edges.
BRTEDG	**$01**	Count bright (low-to-high) edges.

When **DRKPIX** or **BRTPIX** is selected, the command counts either dark pixels or bright pixels, depending on which modifier is used. Here's an example:

```
OWOUT owio, 0, [CNTNEW|DRKPIX, 32, 64]
GOSUB Ready
```

This will count all the dark pixels between locations 32 and 64, inclusive. Suppose the binary pixel array looked like this, where vertical bars represent the location 32 and 64 boundaries:

… 0000111000	11111111000000001111110000111111	1111100000 …
31 32		64 65

In this case, there would be twelve dark pixels counted, and a **12** would be written at the next available byte location in the results buffer. Assuming that this **CNTNEW** was the first command after an image acquisition, that would be location **RESULTS + 5** (**$25**), so we could read the result into the Byte variable **dark_count** with the following code:

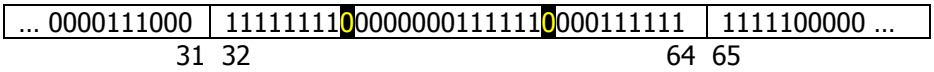
```
OWOUT owio, 0, [DUMPADR, RESULTS + 5]
OWIN owio, 2, [dark_count]
```

Once this sequence of code has executed, remember, the results buffer pointer is reset to **RESULTS** (**$20**), and the next result computed will be buffered there. This is due to the reset sent at the end of the **OWIN** statement.

When **DRKEDG** or **BRTEDG** is selected, the command looks for the first pixel that does *not* match the selected intensity (i.e. **DRK** or **BRT**), starting from the Left limit. It then looks for the next pixel that *does* match. This transition is an edge, and is counted as such. This process continues, accumulating the edge count until the Right limit is reached. Here is an example:

```
OWOUT owio, 0, [CNTNEW|DRKEDG, 32, 64]
GOSUB Ready
```

Here are the same pixels, but with the counted edges highlighted :



There are two such edges within the region of interest, so the result of this command is **2**, which is then buffered at the next available buffer location. Assuming again that this is just a continuation of the code that went before, we could then read this result into the Byte variable **edge_count** from location **$20** (**RESULTS**):

```
OWOUT owio, 0, [DUMPADR, RESULTS]
OWIN owio, 2, [edge_count]
```

But if we're interested in *both* the pixel count and the edge count, it's much more efficient to compute them both, *then* read the results. This is what the code would look like in that case:

```
OWOUT owio, 0, [CNTNEW|DRKPIX, 32, 64]
GOSUB Ready
OWOUT owio, 0, [CNTNXT|DRKEDG]
GOSUB Ready
OWOUT owio, 0, [DUMPADR, RESULTS + 5]
OWIN owio, 2, [dark_count, edge_count]
```

Note that, because these commands are executed immediately, we need to wait for each one to complete by calling **Ready** before sending another. Also note that **edge_count** is now being read from location **RESULTS + 6** (**$26**) instead of **RESULTS** (**$20**). This is because there is no reset this time between reading **dark_count** and **edge_count**.

Also note that **CNTNEW** leaves the internal pointers set to **Begin** and **End**, and **CNTNXT** leaves the internal pointers where they were when it was invoked. For that reason, the second count can use **CNTNXT**, since the **32** and **64** are already established.

In the section that discusses buffered commands, we shall see how this whole sequence can be made yet more efficient by chaining the commands in the command buffer and executing them as a single command.

Locating Pixels and Edges

In addition to counting pixels and edges, it's also useful to know where certain pixels and edges are located within the image. We may want to locate the edge of a web on a paper machine, for example, to make sure it's tracking right, or follow a seam for welding two pipe sections together, or determine the level of liquid in a clear bottle. The commands that do all this are **FNDNEW** and **FNDNXT**:

FNDNEW|Modifiers, Begin, End
FNDNXT|Modifiers

FNDNEW is a constant defined in the PBASIC code template as **$F8** and finds pixels or edges between pixels **Begin** and **End**, inclusive. **Begin** and **End** can range from **1** to **255**, and **Begin** should be less than or equal to **End**. (Remember that binary pixel locations are one-based, not zero-based. The first pixel is pixel #1.)

FNDNXT finds pixels or edges between the current internal Left and Right limits, whatever they might be. These limits are set by the following actions, whichever occurred most recently:

- Any acquire command, which resets Left to **1** and Right to **255**.
- Any **CNTNEW** command. Left ends up at **Begin**; Right, at **End**.
- A **FNDNEW** command. Left is initialized to **Begin**; Right, to **End**.
- Any **FNDNEW** or **FNDNXT** command, wherein Left is moved to the found pixel or edge (searching forward), or Right is moved to the found pixel or edge (searching backward). If the desired feature is not found, Left will equal Right + 1, causing further counts and finds to return zero.

The modifiers that can be used with (i.e. ORed to) **FNDNEW** and **FNDNXT** are:

Name	Value	Description
FWD	**$00**	Search from left-to-right.
BKWD	**$04**	Search from right-to-left.
DRKPIX	**$00**	Locate dark pixels.
BRTPIX	**$02**	Locate bright pixels.
DRKEDG	**$03**	Locate dark (high-to-low) edges.
BRTEDG	**$01**	Locate bright (low-to-high) edges.

These are the same modifiers used with **CNTNEW** and **CNTNXT**, but with two additions: **FWD** and **BKWD**. With these modifiers, you can select which end to start the search from. Searching forward starts from the Left limit and scans towards the right until either the desired feature is found or the Right limit is reached. Searching backwards starts at the Right limit and scans to the left until either the desired feature is found or the Left limit is reached.

Each invocation of **FNDNEW** or **FNDNXT** appends one byte to the results buffer. If the sought-after pixel or edge was found, this byte will be its position (**1** to **255**) in the binary image. If it wasn't found, the result will be **0**.

Another side effect of the find commands is that the Left or Right limit, whichever one you started from, is replaced by the result of the find. So, for example, if the first dark pixel, scanning from the left, were found at position 45, then the new value of Left would become 45. That way, the next time you use **FNDNXT** to scan from the left, the search picks up where the last one ended, at position 45. This makes it possible to chain multiple finds to locate, say, the third occurrence of a certain feature, rather than just the first one. If the result of the search is **0**, the internal Left and Right limits will have crossed, forcing all subsequent searches to result in **0** as well, until these limits are reset to new values.

When the **DRKPIX** or **BRTPIX** modifier is selected, **FNDNEW** and **FNDNXT** will look for the first dark or light pixel in the selected direction, depending on the modifier used. For example, suppose we want to find the first dark pixel between locations 32 and 64, scanning from left to right. Here's the code that does it:

```
OWOUT owio, 0, [FNDNEW|FWD|DRKPIX, 32, 64]
GOSUB Ready
```

As with the count commands, we need to call **Ready** before issuing further commands. Now suppose the binary pixel array looked like this, where vertical bars represent the location 32 and 64 boundaries:

… 0000111000	11111111000000001111110000111111	1111100000 …
	31 32 64 65	

The result of the **FNDNEW** would thus be **40**, the location of the first dark pixel, starting from location **32** and moving right. That value would be appended to the results buffer, as with the count commands, and we can read it into the Byte variable **location** like this (assuming this is the first command after the last acquisition:

```
OWOUT owio, 0, [DUMPADR, RESULTS + 5]
OWIN owio, 2, [location]
```

Now suppose we wanted to locate the first dark-to-light edge in the same region. Here's the code:

```
OWOUT owio, 0, [FNDNEW|FWD|BRTEDG, 32, 64]
GOSUB Ready
```

We use **BRTEDG** here because that's the kind of edge (low-to-high) **FNDNEW** has to find. But for there to exist such an edge within the given range, there has to be a dark pixel first, followed by a light one. So, given the same image as above, we find our edge at location **48**, as highlighted:

… 0000111000	11111111000000001111110000111111	1111100000 …
	31 32 64 65	

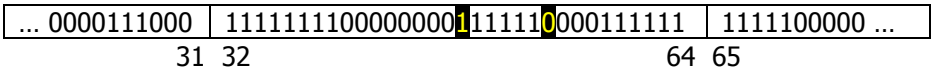
Now, suppose we want to find the first bright *object* in the same region. A bright object is one that begins with a low-to-high transition and ends with a high-to-low transition. Here is where chaining two finds comes in handy:

```
lft_edge VAR Byte
rgt_edge VAR Byte

OWOUT owio, 0, [SETEXP, 60, SETBIN, 100, 3, 0, ACQBIN]
GOSUB Ready

OWOUT owio, 0, [FNDNEW|FWD|BRTEDG, 32, 64] '
GOSUB Ready                                 '___ Chained finds
OWOUT owio, 0, [FNDNXT|FWD|DRKEDG]          '
GOSUB Ready                                 '

OWOUT owio, 0, [DUMPADR, RESULTS + 5]
OWIN owio, 2, [lft_edge, rgt_edge]
IF (rgt_edge) THEN
  DEBUG "Object found starting at ", DEC lft_edge, " and ending at ", DEC rgt_edge - 1
ELSE
  DEBUG "No object found."
ENDIF
```

There are several things to talk about in the above code:

- First, once we've located the first edge, we want to continue from where we left off to find the second one, so we use **FNDNXT** for the second, leaving off the **Begin** and **End** locations.
- Second, since we've chained two commands in a row, after the **ACQBIN**, our results will be found sequentially, beginning at location **RESULTS + 5**.

- Third, there's a chance that the first edge won't have been found. But we didn't check for that; we just plowed ahead, looking for the second edge. But remember the way find works: if it doesn't find something once, it won't find anything on subsequent **FNDNXT**s either, until the Left and Right limits are reset to new values. So we're safe there.
- Fourth, we're only checking for the presence of the right edge in the **IF** statement to see if the entire object is present. Again, that's because if the left edge wasn't found, the right edge is automatically not found either. So that's all we need to check.
- And finally, what's with the "**– 1**" in the first **DEBUG** statement? Well, here are the two edges we would have found with the above image:

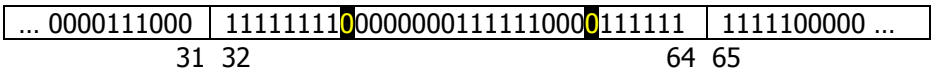


The second edge is located one pixel beyond what we consider to be the right edge of the object so, to point to the right edn of the object we need to subtract one. On the other hand, if we're interested in the *size* of the object, we can just subtract **lft_edge** from **rgt_edge**.

Now, suppose the central portion of above image represents a backlit bagel. In this situation, the bagel consists of two dark, silhouetted areas separated by a bright hole. What we're after here is the diameter of the bagel; we don't care about the hole. This is where a backward search comes into play. First we locate the first bright-to-dark edge scanning forward. Next we locate the first bright-to-dark edge scanning backward. These will be the extreme left and right edges of the bagel, from which we can compute its diameter. Here's the code:

```
lft_edge VAR Byte
rgt_edge VAR Byte

OWOUT owio, 0, [SETEXP, 60, SETBIN, 100, 3, 0, ACQBIN]
GOSUB Ready
OWOUT owio, 0, [FNDNEW|FWD|DRKEDG, 32, 64]
GOSUB Ready
OWOUT owio, 0, [FNDNXT|BKWD|DRKEDG]
GOSUB Ready
OWOUT owio, 0, [DUMPADR, RESULTS + 5]
OWIN owio, 2, [lft_edge, rgt_edge]
IF (rgt_edge) THEN
  DEBUG "Bagel found with diameter ", DEC rgt_edge - lft_edge + 1
ELSE
  DEBUG "No bagel found."
ENDIF
```

Okay, everything looks as expected, except for that "**+ 1**". What's up with that? Here are the edges that the program found:



When scanning backward for an edge, **FNDNXT** looks for the first bright pixel, which is at location 64, then, moving right to left, the first dark pixel after that, which is the one highlighted above and which is part of the bagel itself. Hence, the necessary addition to get the diameter.

Of course, all this assumes that we're looking at the largest part of the bagel, which would be a real coincidence. In a subsequent section, we'll see how to parlay this into an application that inspects bagels moving past on a conveyor and that finds their actual diameters.

Buffering Commands

In the prior section, we saw how commands can be chained in immediate execution mode, and how a call to **Ready** must be performed after some of them. Here we shall see how to eliminate these extraneous calls by buffering a whole sequence of commands. When this is done, one call to **Ready** has to be performed after the commands are executed (regardless of which commands were buffered). But that's it: just a single call to **Ready**.

The buffering commands are as follows:

"<", commands, being, buffered, ">"

The command **"<"** can only be executed in immediate mode, and it puts the TSL1401R driver in buffered mode. Commands entered after that are buffered in the driver's memory, beginning at location **RESULTS + 5**, but are not executed. When the command **">"** is encountered, it is buffered, too, and execution begins from the beginning of the buffer. Each buffered command is executed in turn until the **">"** is reached. Because commands are buffered in what will become the results area of memory, they are likely to get clobbered as results get appended there. However, execution will always be at least one step ahead of the results, so the only commands that get clobbered will be ones that have already executed.

Here are some additional important points:

- When **"<"** is sent in immediate mode, and when **">"** is encountered when executing from the buffer, the internal results pointer is reset to **RESULTS** (**$20**). That means further results will be appended to the buffer beginning at that point.
- Do not buffer the **ACQGRAY**, **DUMPID**, **DUMPFLAGS**, or **DUMPADR**; and do not buffer more than eleven bytes, including the **">"**. Doing so will raise an error condition, and you will need to run the error recovery procedure outlined above, or else reload your program.
- Since nothing gets executed until the **">"** is received, you can send everything up to that point but defer sending the **">"** until the time is right to begin execution.

Now, let's see how to rewrite our code from the previous example to use buffering:

```
lft_edge VAR Byte
rgt_edge VAR Byte

OWOUT owio, 0, [SETEXP, 60, SETBIN, 100, 3, 0, ACQBIN]
GOSUB Ready
OWOUT owio, 0, ["<", FNDNEW|FWD|DRKEDG, 32, 64] '__ Buffered commands.
OWOUT owio, 0, [FNDNXT|BKWD|DRKEDG, ">"]        '
GOSUB Ready
OWOUT owio, 0, [DUMPADR, RESULTS]
OWIN owio, 2, [lft_edge, rgt_edge]
IF (rgt_edge) THEN
  DEBUG "Bagel found with diameter ", DEC rgt_edge - lft_edge + 1
ELSE
  DEBUG "No bagel found."
ENDIF
```

By buffering the finds, we eliminate one call to **Ready**, since the driver firmware waits until *all* the buffered commands have executed before signaling that it is no longer busy. Notice, too, that we now have to read our results from address **RESULTS**, instead of **RESULTS + 5**. This is because the **"<"** reset the results pointer back to that point after the **ACQBIN** executed. How many bytes did we actually buffer, anyway? In the first line, there are three: the **"<"** doesn't get buffered. In the second line there

are two: the **">"** *does* get buffered. That's five altogether, so we're well within the eleven limit. As we shall see in the next section, we can even include an acquisition command in the buffer.

Bagels and Bottles: Putting it All Together

Let's integrate everything we've learned now into a real application. In this application, we've got bagels passing by single-file atop a black conveyor belt. They are being lighted from above, so they will look bright against a dark background. We want to record the outer diameter of each one. For this app, we assume that the bagels are round and not oval. We also assume that there is a wide enough gap between each pair of bagels that we will see it at least once. Finally, we assume that the conveyor spans the entire field of view and that there are no crumbs on it to confuse the camera. (In real life, we would have to question *every one* of these assumptions!) Here is the meat of the code that will do the work. As with all previous examples, we must wrap it in the template given at the end of this chapter to be complete:

```
 1  lft_edge VAR Byte
 2  rgt_edge VAR Byte
 3  max_dia  VAR Byte
 4
 5  OWOUT owio, 1, [SETEXP, 30]
 6  OWOUT owio, 0, [SETBIN, 128, 10, FIXED|LEVEL]
 7
 8  DO
 9    OWOUT owio, 0, ["<", ACQBIN, FNDNXT|FWD|BRTEDG]
10    OWOUT owio, 0, [FNDNXT|BKWD|BRTEDG, ">"]
11    GOSUB Ready
12    OWOUT owio, 0, [DUMPADR, RESULTS + 5]
13    OWIN owio, 2, [lft_edge, rgt_edge]
14    IF (rgt_edge) THEN
15      max_dia = rgt_edge - lft_edge + 1 MIN max_dia
16    ELSEIF (max_dia) THEN
17      DEBUG "Bagel diameter: ", DEC max_dia, CR
18      max_dia = 0
19    ENDIF
20  LOOP
```

Line numbers have been added to the left of the actual program so we can discuss each line here:

Line	Description
1-3	Some variables that aren't included in the template are defined here. You're already familiar with **lft_edge** and **rgt_edge**. The variable **max_dia** is used to keep track of the largest diameter seen so far on a particular bagel.
5	Exposure time is set to 30.
6	Binary acquisition parameters are set: Threshold = 128; Hysteresis = 10; Threshold is fixed instead of floating, and comparisons are done on the level instead of a window.
8	Once the basic parameters have been established, they don't change. The real work can now commence within a **DO** loop.
9	Here, we buffer the plain binary acquire command, and the command that looks for the first dark-to-bright edge. Notice that we just use **FNDNXT** here, without the **Begin** and **End** limits. This is because we're scanning the entire field of view. When a new image is acquired, these limits are automatically reset to **1** and **255**, respectively.
10	Continuing with the buffering, we include the command that looks for the same kind of edge, but coming in from the right. We also end the buffering, which starts the whole chain of commands executing.
11	Here, we wait for all the commands to finish executing. When they do, the image will have been acquired, and both edges will have been located – all with only four bytes in the buffer.
12	Now we're ready to read the results, so start reading from location **RESULTS + 5**. This is because the **ACQBIN** is buffered along with the finds, and it always adds five bytes to the results buffer, thus pushing the find results five bytes higher.
13	Read the positions of the left and right edges.
14	If we found the right edge, the left edge is there, too.
15	The current diameter is **rgt_edge – lft_edge + 1**. If that's greater than the maximum diameter seen so far on this bagel, make it the maximum diameter, using the **MIN** operator.
16	If we didn't see an edge, we're between bagels, and if **max_dia** is non-zero, one has *just* gone past that we haven't recorded yet.
17	So tell the world what we just saw.
18	Reset **max_dia** to zero, so it's ready for the next bagel.
20	And back for another scan.

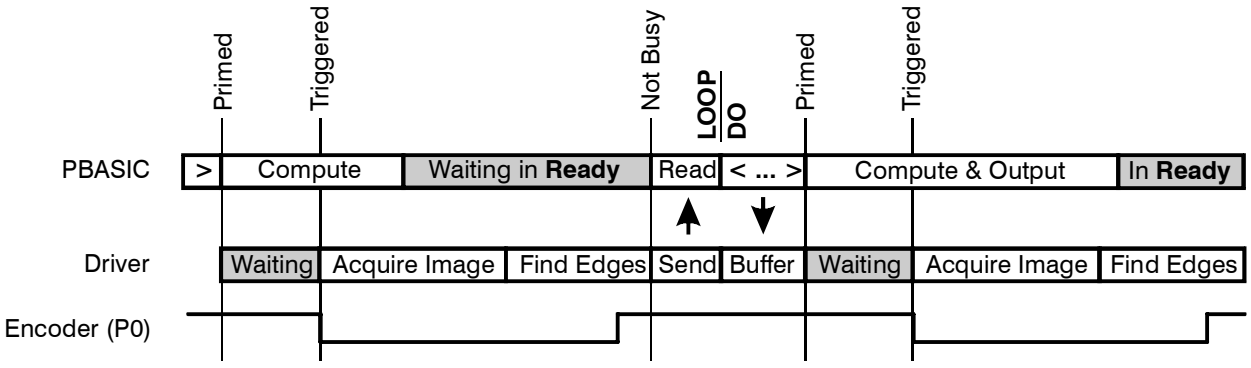
So that's it: a complete inspection-and-reporting program in 20 lines of code. But let's take it one step further. One technique where buffering really shines is when it's combined with externally-triggered exposures. This makes it possible to buffer an entire acquisition-and-analysis sequence *and to begin execution immediately*. But nothing will happen until a falling edge on **P3** is detected, whereupon the driver firmware springs into action by itself, "snaps the picture", and does the analysis. All the PBASIC program has to do is call **Ready** when *it's* ready, to see if new results are available. In fact, if further processing of the results read from the firmware is required, we can even "arm" the firmware ahead of that processing, in order to overlap processing one image with acquiring the next.

To demonstrate this technique, we're going to change the mission slightly. Instead of looking for each bagel's maximum width, we're going to measure each one to compute the overall area covered by the bagel, including the hole. For measurement consistency, we've installed an encoder on our hypothetical conveyor that issues a pulse every quarter inch of travel. So, if we measure the overall width of each bagel every quarter of an inch of travel and accumulate the sum of those widths, we will have measured its area once it has passed.

Here's one way to do it:

```
 1  lft_edge VAR Byte
 2  rgt_edge VAR Byte
 3  area     VAR Word
 4
 5
 6  OWOUT owio, 1, [SETEXP, 30]
 7  OWOUT owio, 0, [SETBIN, 128, 10, FIXED|LEVEL]
 8
 9  DO
10    OWOUT owio, 0, ["<", ACQBIN|XTRIG, FNDNXT|FWD|BRTEDG]
11    OWOUT owio, 0, [FNDNXT|BKWD|BRTEDG, ">"]
12    IF (rgt_edge) THEN
13      area = area + rgt_edge - lft_edge + 1
14    ELSEIF (area) THEN
15      DEBUG "Bagel area: ", DEC area, CR
16      area = 0
17    ENDIF
18    GOSUB Ready
19    OWOUT owio, 0, [DUMPADR, RESULTS + 5]
20    OWIN owio, 2, [lft_edge, rgt_edge]
21  LOOP
```

This is very similar to the previous example, except that the statement order has been changed. We've buffered the acquire and finds in the loop first, but then look what happens: we start computing with data that hasn't yet been read. In PBASIC, all variables are initialized to zero; So the first time through the loop, neither the **IF** nor the **ELSEIF** conditions will be true, and that section, between lines 12 and 17, will simply be skipped. *Then* comes the call to **Ready**, because right then we want to read some new data, which we do on lines 19 and 20. Now, when we loop back, we can prime the next exposure right away, *and then* get on with our area calculations and, possibly, output. This has the effect of overlapping the coprocessor's work with the PBASIC program's in the most efficient manner possible. Of course, in any real application, you also have to make sure there's enough time between encoder pulses to get everything done! The illustration below shows the timeline of events:



In the monitor program section, we looked at bottles – some full, some not-so full, some with caps, and some without. What we want to do is pass the bottle if all of the following conditions prevail:

1. The liquid level is detected between pixels 111 and 129, and
2. The cap is detected, and it has a size of at least 36 pixels.
3. The top of the cap is no higher than pixel 205 (i.e. it's pushed on all the way).

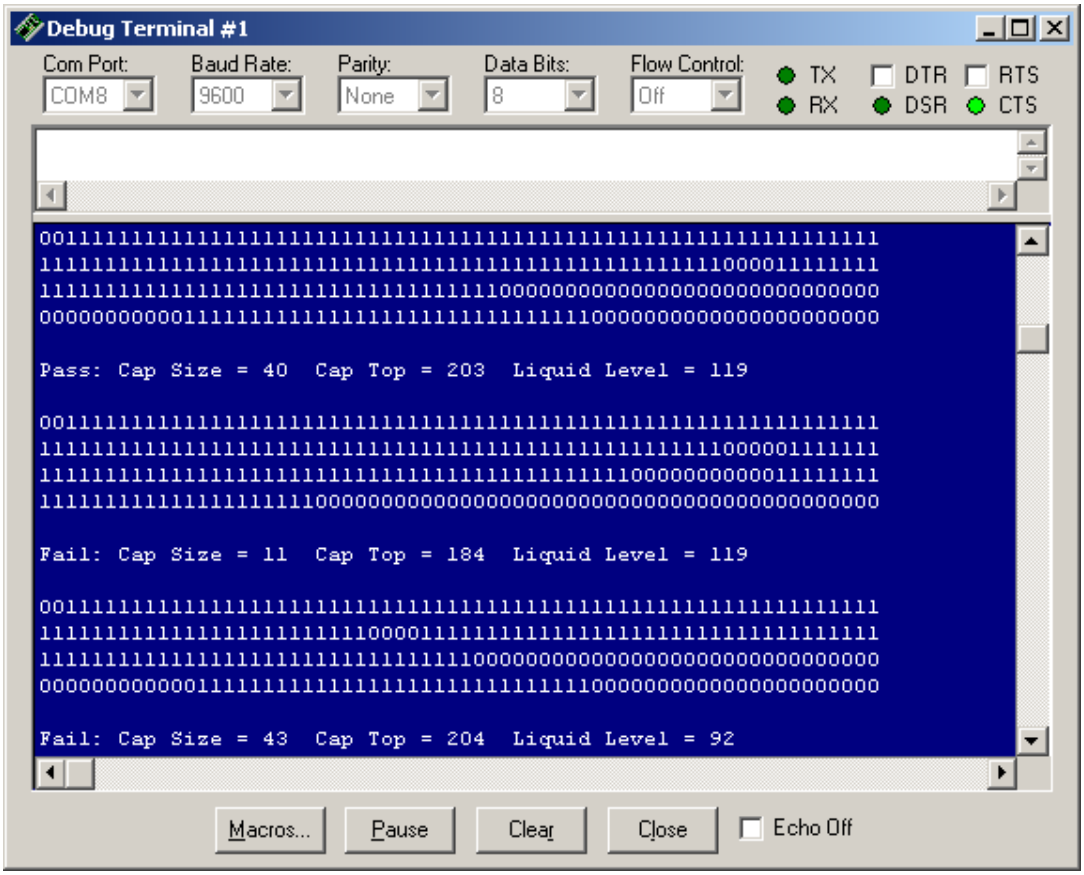
Here is the program that sorts this all out and gives a pass/fail grade to each bottle it sees:

```
liq_btm VAR Byte
liq_top VAR Byte
cap_btm VAR Byte
cap_top VAR Byte
liq_lvl VAR Byte
cap_siz VAR Byte
i       VAR Byte
pix     VAR Byte

OWOUT owio, 1, [SETEXP, 60]
OWOUT owio, 0, [SETBIN, 0, 1, FLOAT|LEVEL|5]
DO
  SERIN 16, 84, [WAIT(" ")]
  OWOUT owio, 0, ["<", ACQBIN, FNDNXT|FWD|DRKEDG]
  OWOUT owio, 0, [FNDNXT|FWD|BRTEDG, FNDNXT|BKWD|DRKEDG]
  OWOUT owio, 0, [FNDNXT|BKWD|BRTEDG, ">"]
  GOSUB Ready
  OWOUT owio, 0, [DUMPADR, PIXELS]
  DEBUG CLS
  FOR i = 0 TO 31
    OWIN owio, 0, [pix]
    DEBUG BIN8 pix REV 8
    IF (i & 7 = 7) THEN DEBUG CR
  NEXT
  OWOUT owio, 1, [DUMPADR, RESULTS + 5]
  OWIN owio, 2, [liq_btm, liq_top, cap_top, cap_btm]
  cap_siz = cap_top - cap_btm
  liq_lvl = liq_top + liq_btm >> 1
  IF (cap_siz > 35 AND cap_top < 206 AND liq_lvl > 110 AND liq_lvl < 130) THEN
    DEBUG CR, "Pass"
  ELSE
    DEBUG CR, "Fail"
  ENDIF
  DEBUG ": Cap Size = ", DEC cap_siz, "  Cap Top = ", DEC cap_top
  DEBUG "  Liquid Level = ", DEC liq_lvl
LOOP
```

Unlike the prior program, which relies on a falling edge on **P3** to trigger a new scan, this one operates more in demo mode, in that the trigger comes from tapping the spacebar in the DEBUG window. Here's a blow-by-blow description:

Line	Description
1-8	Some variables that aren't included in the template are defined here.
10	Exposure time is set to 60.
11	Binary acquisition parameters are set: Floating threshold = 0; Hysteresis = 1; Filter = 5
12	Beginning of the main program loop.
13	Wait for the spacebar.
14-16	Cue up and execute the commands that snap the picture and find the first dark object edges (liquid meniscus) and the last dark object edges (cap).
17	Wait for everything that's been cued up to finish.
18	Start dumping the binary pixels.
19-24	Display the binary pixels in four lines, beginning two lines down from the previous output.
25	Start dumping the results of the **FNDNXT**s. Do you notice the **1** in the **OWOUT** statement? This sends the reset that terminates the pixel dumping from line 21.
26	Read the results of the **FNDNXT**s.
27	Compute the cap size as the difference between its edge locations.
28	Compute the liquid level as the average of the two edge locations.
29	Test to make sure all the observed locations and sizes meet the specs.
30, 32	Print pass/fail.
34, 35	Print the observed data.

Here is what the output looks like:



Automatic Exposure

For accurate measurements, the only thing that should ever change is the size or position of what we are trying to measure. If a subject isn't in perfect focus or isn't lit with perfect evenness, it can seem to grow and shrink with variations in lighting. And despite our best efforts, lighting isn't always as controlled as we'd like it to be. So we often need to compensate by adjusting the exposure time to changing light levels. In cases where a bright object is always within the field of view and/or light intensity changes very slowly, this is pretty easy.

The image acquisition commands all record the intensity of the brightest pixel. This information can be used to maintain a constant maximum brightness under varying light conditions. The simplest rule is this:

1. If the maximum brightness is greater than 220, we decrease the exposure time by one.
2. If the maximum brightness is less than 200, we increase the exposure time by one.
3. If the maximum brightness is between 200 and 220, we leave the exposure time alone.

We want to keep the brightness high, for maximum analog resolution, but we don't want it to saturate. When brightness levels reach 255, that means the limit of the TSL1401R's voltage output has been reached, so there's no way to tell if the actual brightness might have been higher than that. So we try to keep the maximum brightness between 200 and 220. Here's a snippet of code that illustrates the rule in action. It uses the LEDs on the MoBoStamp-pe to indicate the current light level: Red is too high; green is too low; yellow (red and green together) is just right.

```
red      PIN 13                        'Pin for red LED on MoBoStamp-pe.
green    PIN 14                        'Pin for green LED on MoBoStamp-pe.

max_brt  VAR Byte                      'Maximum brightness read from driver.
exp_time VAR Byte                      'Current exposure time.

exp_time = 30                          'Establish initial exposure time,

OWOUT owio, 1, [SETEXP, exp_time]      '  and set it.

DO                                     'Do repeatedly:
  OWOUT owio, 0, [ACQBIN]              '  Acquire an image.
  GOSUB Ready                          '  Wait for not-busy state.
  OWOUT owio, 0, [DUMPADR, MAXPIX]     '  Read the maximum pixel value.
  OWIN owio, 2, [max_brt]
  IF (max_brt < 200) THEN              '  Is it less than 200?
    exp_time = exp_time + 1 MAX 255    '    Yes: Increment exposure time (to 255 max),
    OWOUT owio, 0, [SETEXP, exp_time]  '           and set it.
    LOW green : HIGH red               '         Indicate as green (too low).
  ELSEIF (max_brt > 220) THEN          '  Is it greater than 220?
    exp_time = exp_time - 1 MIN 1      '    Yes: Decrement exposure time (to 1 min),
    OWOUT owio, 0, [SETEXP, exp_time]  '           and set it.
    LOW red : HIGH green               '         Indicate as red (too high).
  ELSE                                 '  Is it between these values?
    LOW green : LOW red                '    Yes: Indicate as yellow (just right).
  ENDIF
LOOP
```

Keeping the maximum brightness at a constant level is *all* this code does, by the way. One thing to note is that brightness is being measured continuously here. In applications where images are acquired sporadically, this approach may not work, unless acquisitions are performed between them just to measure image brightness. If you run this program, you will notice that it responds to changes in brightness rather slowly. In situations where brightness can vary faster than simple incrementing or decrementing can compensate for, it may be necessary to adjust by an amount proportional to the

difference between the desired and actual levels to get a faster response. For example, the auto-exposure method used in the TSL1401 Monitor program is:

```
exp = $E000 / (maxbrt / $FF + 1 * maxbrt) */ exp max 255 min 1
```

where **exp** is the exposure time, and **maxbrt** is the maximum brightness read from location **MAXPIX** after each acquisition. This method responds instantly to changes in maximum brightness, which may not always be a good thing – especially when those changes occur because of changes in the subject and not changes in the lighting. But this just illustrates that there are many possible approaches and that you have to pick one appropriate to your individual application.

PBASIC Code Template

Here is the code template, **TSL1401_template.bpe**, which defines all the constants and subroutines used by the examples above. You can also download it from the Parallax website to use with these examples and with your own programs.

```
' =============================================================================
'
'   File...... tsl1401_template.bpe
'   Purpose... Code template for the TSL1401-DB driver firmware.
'   Author.... Parallax, Inc.
'   E-mail.... support@parallax.com
'   Started... 20 July 2007
'   Updated...
'
'   {$STAMP BS2pe}
'   {$PBASIC 2.5}
'
' =============================================================================


' -----[ Program Description ]---------------------------------------------

' This is a blank template used for interacting with the TSL1401R driver
' firmware in the MoBoStamp-pe's AVR coprocessor.

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

owio            PIN     6     'Pin for OWIN and OWOUT to AVR coprocessor.

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

' Commands

SETLED          CON     $EB   'Set LED strobe and brightness/time from next byte.

  'Flag to OR to brightness/time (0 - 127) value.

  TIME          CON     $80   'Set strobe: value (0 - 127) is 0 - 3.4mS at 100% on.
  INTEN         CON     $00   'Set intensity: value (0 - 127) is 0 - 49.6% on.

SETBIN          CON     $EC   'Set threshold, hysteresis, and filter (3 bytes).

  'Filter flags, ORed with filter value, NOT with SETBIN.

  FLOAT         CON     $80   'Threshold is floating per filter value (0 - 7).
  FIXED         CON     $00   'Threshold is fixed.
  WINDOW        CON     $40   'Threshold is a window (outside of hysteresis band).
  LEVEL         CON     $00   'Threshold is a level with hysteresis.
```

```
SETEXP          CON     $EE   'Set exposure to byte (1 - 255) following: 0.27 - 68mS.

ACQGRAY         CON     $A0   'Acquire and dump a grayscale image.
ACQBIN          CON     $A4   'Acquire a binary image.
ACQAND          CON     $A1   'Acquire binary image ANDed w/ previous.
ACQOR           CON     $A2   'Acquire binary image ORed w/ previous.
ACQXOR          CON     $A3   'Acquire binary image XORed w/ previous.
ACQANDNOT       CON     $A5   'Acquire binary image ANDed w/ NOT prev.
ACQORNOT        CON     $A6   'Acquire binary image ORed w/ NOT prev.
ACQXORNOT       CON     $A7   'Acquire binary image XORed w/ NOT prev.

ACQDIFF         CON     $A3   'Idiom for ACQXOR.
ACQSAME         CON     $A7   'Idiom for ACQXORNOT.

  XTRIG         CON     $08   'External trigger flag, ORed to ACQ commands.

CNTNEW          CON     $C8   'Count pixels/edges between new bounds.
CNTNXT          CON     $C0   'Count pixels/edges between current bounds.
FNDNEW          CON     $F8   'Find first pixel/edge between new bounds.
FNDNXT          CON     $F0   'Find first pixel/edge between current bounds.

  'Modifiers, ORed to CNTNEW, CNTNXT, FNDNEW, and FNDNXT.

  NXT           CON     $00   'Continue from where last CNT or FND left off.
  BKWD          CON     $04   'Search backward.
  FWD           CON     $00   'Search forward.
  DRKPIX        CON     $00   'Target is a dark pixel.
  BRTPIX        CON     $02   'Target is a bright pixel.
  DRKEDG        CON     $03   'Target is a bright-to-dark edge.
  BRTEDG        CON     $01   'Target is a dark-to-bright edge.

DUMPADR         CON     $DA   'Dump data, beginning at addr, and until reset.

  'Address constants for single byte arg following DUMPADR.

  PIXELS        CON     $00   'Beginning of binary pixel buffer (32 bytes).
  RESULTS       CON     $20   'Beginning of results buffer.
  MINPIX        CON     $20   'Value of darkest pixel (0 - 255).
  MINLOC        CON     $21   'Location of darkest pixel (0 - 127).
  MAXPIX        CON     $22   'Value of brightest pixel (0 - 255).
  MAXLOC        CON     $23   'Location of brightest pixel (0 - 127).
  AVGPIX        CON     $24   'Average pixel value (0 - 255).

DUMPID          CON     $DD   'Dump the firmware ID (returns 3 bytes).

DUMPFLAGS       CON     $DF   'Dump error flags (returns 1 byte).

  'Bit positions in returned byte.

  BADCMD        CON     $80   'Unrecognized command.
  CANTBUF       CON     $40   'Attempt to buffer unbufferable command.
  CMDOVF        CON     $20   'Command buffer overflow.
  DATOVF        CON     $10   'Result data buffer overflow.

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

flags           VAR     Byte
busy            VAR     Bit

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

PAUSE 10                      'Wait for AVR to finish reset.
```

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

' Your program code goes here.

END

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

' -----[ Ready ]------------------------------------------------------------
' Wait for the driver to become not busy.

Ready:
  DO
    OWIN owio, 4, [busy]     'Read busy bit.
  LOOP WHILE busy            'Keep checking until it goes low.
  RETURN

' -----[ GetError ]--------------------------------------------------------
' Read the error flags from the driver.

GetError:
  OWOUT owio, 0, [DUMPFLAGS]   'Read the error flags.
  OWIN owio, 0, [flags]
  IF (flags = $FF) THEN        'If $FF, driver is waiting for a reset.
    OWOUT owio, 1, [DUMPFLAGS] 'So reset and try again.
    OWIN owio, 0, [flags]
  ENDIF
  RETURN
```

Command Summary

Name	Val.	Description	Buf-Fer[1]	Busy[2]	Modifiers[3]	Args	Re-sults[4]
DUMPID	**$DD**	Dumps two-letter ID and version byte.	No	No	None	None	0
DUMPFLAGS	**$DF**	Dumps single error byte.	No	No	None	None	0
DUMPADR	**$DA**	Dumps memory beginning at **Addr**, until reset.	No	No	None	**Addr**	0
SETEXP	**$EE**	Set exposure time to **ExpTime**.	Yes	No	None	**Exp**	0
SETBIN	**$EC**	Set binary acquisition parameters.	Yes	No	None	**Thld, Hyst, Mode**	0
SETLED	**$EB**	Set LED brightness (**INTEN|Value**) or strobe time (**TIME|Value**).	Yes	No	**INTEN ($00)**[5] **TIME ($80)**[5]	**Value**	0
ACQGRAY	**$A0**	Acquire binary image; dump gray values.	No	Yes	**XTRIG($08)**	None	5
ACQBIN	**$A4**	Acquire binary image.	Yes				
ACQAND	**$A1**	Acquire and AND new binary image to old image.					
ACQOR	**$A2**	Acquire and OR new binary image to old image.					
ACQXOR ACQDIFF	**$A3**	Acquire and XOR new binary image to old image.					
ACQNOTAND	**$A5**	Acquire and AND new binary image to NOT old image.					
ACQNOTOR	**$A6**	Acquire and OR new binary image to NOT old image.					
ACQXORNOT ACQSAME	**$A7**	Acquire and XOR new binary image to NOT old image.					
CNTNEW	**$C8**	Count pixels/edges between new limits.	Yes	Yes	**FWD ($00)**[6] **BKWD ($04)**[6] **DRKPIX ($00)** **BRTPIX ($02)** **DRKEDG ($00)** **BRTEDG ($01)**	**Begin, End**	1
CNTNXT	**$C0**	Count pixels/edges between current limits.				None	
FNDNEW	**$F8**	Find pixels/edges between new limits.				**Begin, End**	
FNDNXT	**$F0**	Find pixels/edges between current limits.				None	
"<"	**$3C**	Begin buffering commands.	No	No	None	None	0
">"	**$3E**	End buffering, execute buffer, then enter immediate mode.	Yes[7]	Yes	None	None	0 – 16[8]

Notes:

1. The **Buffer** column indicates whether the command can be buffered.
2. The **Busy** column indicates whether the busy bit needs to be read as zero after the command is sent and before further interaction with the firmware can take place. (Applies only to immediate mode.)
3. **Modifiers** are ORed to the command byte, except where noted. Modifiers with a value of zero (**$00**) may be omitted; however including them can make a program more readable.
4. The **Results** column indicates how many bytes are appended by the command to the results buffer.
5. **INTEN** and **TIME** modify the **Value** parameter, not the command itself.
6. **FWD** and **BKWD** apply to **FNDNEW** and **FNDNXT** only, not to the count routines.
7. The **">"** command can *only* be buffered. It cannot be used in immediate mode.
8. The number of results produced by a buffered sequence depends on the commands in the buffer and will be the sum of what the individual commands produce.

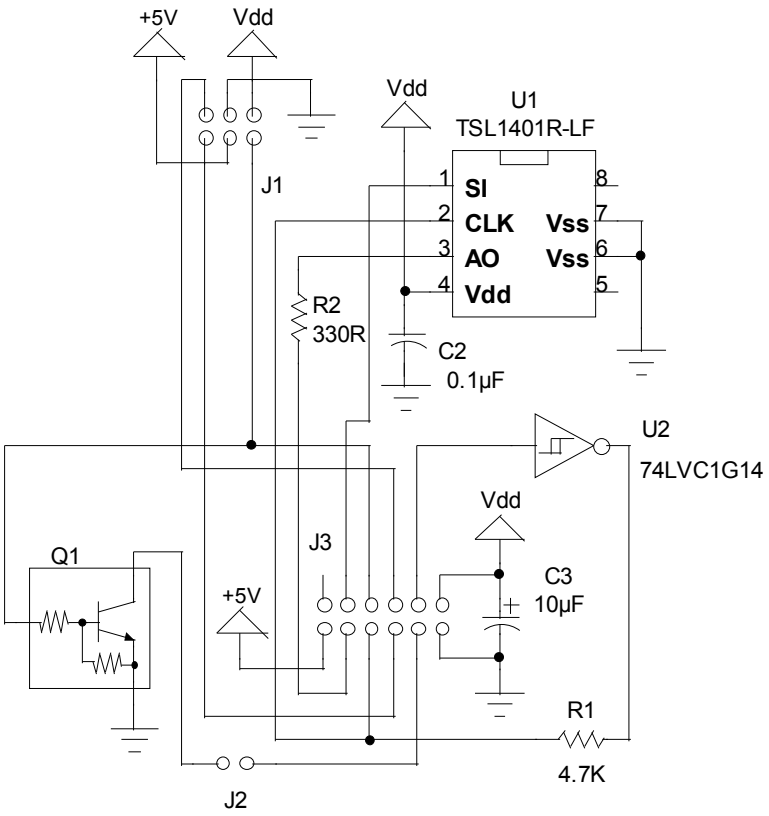
Lighting

No discussion of machine vision would be complete without an entire chapter on lighting. The fact is that any vision application will succeed or fail based on how well you are able to control the lighting. If you try to use ambient lighting, you will almost surely fail in your endeavor. It really is that simple.

Fortunately, there are ample resources on the internet that cover this important topic. A few of the better ones can be found at these URLs:

- Melles Griot: http://www.mellesgriot.com/products/machinevision/lif_1.htm
- Advanced Illumination: http://advill.com/uploads/downloads/A%20Practical%20Guide%20to%20Machine%20Vision%20Lighting.pdf
- Edmund Scientific: http://www.edmundoptics.com/techSupport/DisplayArticle.cfm?articleid=264
- *Vision & Sensors* Magazine: http://www.visionsensorsmag.com/CDA/Articles/Cover_Story/BNP_GUID_9-5-2006_A_10000000000000097315

Schematic



Index