```
    IF (pixels(i) = which) THEN
      sum = sum + i
      cnt = cnt + 1
    ENDIF
  NEXT
  IF (cnt) THEN sum = sum / cnt
  RETURN
```

To use this program, set the values for **which** and **line** according to the color and size of your tape. Unscrew the lens until there's a 3mm gap between the underside of the lens bezel and the top of the lens holder. Place the BOE-Bot on the line, and then turn it on. It should be able to follow the line quite smoothly, strobing the LEDs as it goes. If it wanders off the line, you may need to readjust the **line** constant, or you may just not have enough contrast between the line and your floor.

Operation with a Parallax Motherboard

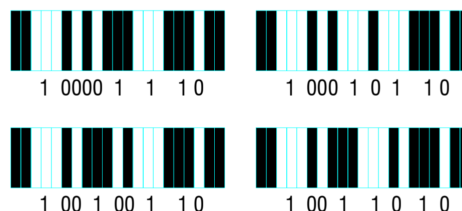
When used in conjunction with a Parallax motherboard, operation of the StrobeLED-DBM takes place automatically with the provided software/firmware. Consult either the TSL1401-DB or the PropCAM-DB manual for details on how to set the strobe intensity and timing.

One possible application for the StrobeLED in this context is to illuminate barcodes for a TSL1401-DB-centric barcode reader. Since the resolution of the TSL1401R is limited, we shouldn't expect to read UPC codes; but it's simple enough to create our own, simpler barcodes to read and identify. A barcode consists of black bars separated by white bars, each of varying widths. There are always an odd number of bars, since both outside bars have to be black. The relative widths of the bars determine the code that the barcode reader reads. It's always nice to know when a barcode is being read upside down, so the directionality needs to be encoded as well. Finally, to keep the overall width of each barcode constant, the number of wide bars and the number of narrow bars has to be kept constant over the entire range of codes.

Let's create a barcode defined as follows:

- At each end will be black bar two units wide.
- The second bar will be a white bar three units wide.
- The second-to-last bar will be a white bar one unit wide.
- In between will be a code area consisting of four black bars and three white bars, each either three units wide or one unit wide, such that there are always four narrow bars and three wide bars.
- Wide bars will be assigned a binary "1"; narrow bars, a "0".

With this definition, every barcode will total 2 + 3 + (4 × 1) + (3 × 3) + 1 + 2 = 21 units wide. The bars on the end give us a threshold width with which to compare the other bars. Anything wider is a "1"; narrower, a "0". So the codes we read will look like these four examples:



Notice that the first bit is always a "1" and the last bit a "0". These correspond to the second bar and second-to-last bar, which are always wide and narrow, respectively. If we read a code where this situation is reversed, we will know it's upside down, so we can reverse the bits.

Because the actual code consists of seven bits, four of which have to be zeroes, there are 7! / (4! × 3!) = 35 possible combinations. All 35, with their binary equivalents, appear at the end of this section. These can be printed out and used to test the barcode reader.

To read a barcode, we shall first assume (for pedantic purposes) that nothing but the barcode we want to read appears in the field of view. In the real world, this is almost never the case, and most barcode specs include a figure for "guard bands", which are the minimum white areas which must appear on either side of the barcode to render it readable. Eventually, to make our code useful, one would want to include guard band detection, so extraneous stuff could be included in the field of view. But for right now, it complicates things more than it is useful to consider.

Every barcode in our definition has twelve edges, starting with a light-to-dark transition. This amounts to six dark edges and six bright edges. So this is the first thing we will check to see if a barcode is valid. Here is the total list of things to check:

- Total number of dark edges and bright edges (must be six apiece).
- Sizes of first and last bars (must be roughly equal).
- Leading bit of nine (if **0**, reverse the bits).
- First and last bits (must now be **1** and **0**, respectively).
- Number of **1** bits, not including the leading **1** (three).
- Resulting code is different from previous code (to eliminate extraneous beeps).

Here is the meat of the PBASIC that performs the barcode reading. It is inserted into the "Program Code" space of the file "TSL1401_template.bpe", which can be downloaded from the Parallax website.

```
 1  cnta            VAR     Byte  'General holders for counts and widths.
 2  cntb            VAR     Byte
 3  edga            VAR     Byte  'Edge location variables.
 4  edgb            VAR     Byte
 5  edgc            VAR     Byte
 6  edgd            VAR     Byte
 7  code            VAR     Word  'Holds the result code.
 8  pcode           VAR     Word  'Holds the previous result code.
 9  exp             VAR     Byte  'Current exposure time.
10  maxbrt          VAR     code  'Maximum brightness measured by firmware.
11
12  exp = 30
13  OWOUT owio, 1, [SETEXP, exp, SETBIN, 128, 5, FIXED, SETLED, 127]
14
15  DO
16    OWOUT owio, 1, ["<", ACQBIN, CNTNXT|DRKEDG, CNTNXT|BRTEDG,
17      FNDNXT|FWD|DRKEDG, FNDNXT|FWD|BRTEDG,
18      FNDNXT|BKWD|DRKEDG, FNDNXT|BKWD|BRTEDG, ">"]
19    GOSUB Ready
20    OWOUT owio, 0, [DUMPADR, MAXPIX]
21    OWIN owio, 2, [maxbrt, cnta, cnta, cnta, cntb, edga, edgb, edgd, edgc]
22    exp = $E000 / (maxbrt / $FF + 1 * maxbrt) */ exp MAX 255 MIN 1
23    OWOUT owio, 0, [$EE, exp]
24    IF (cnta = 6 AND cntb = 6 AND ABS(edgb - edga - (edgd - edgc)) < 5) THEN
25      cnta = edgb - edga + edgd - edgc >> 1
26      code = 0
```

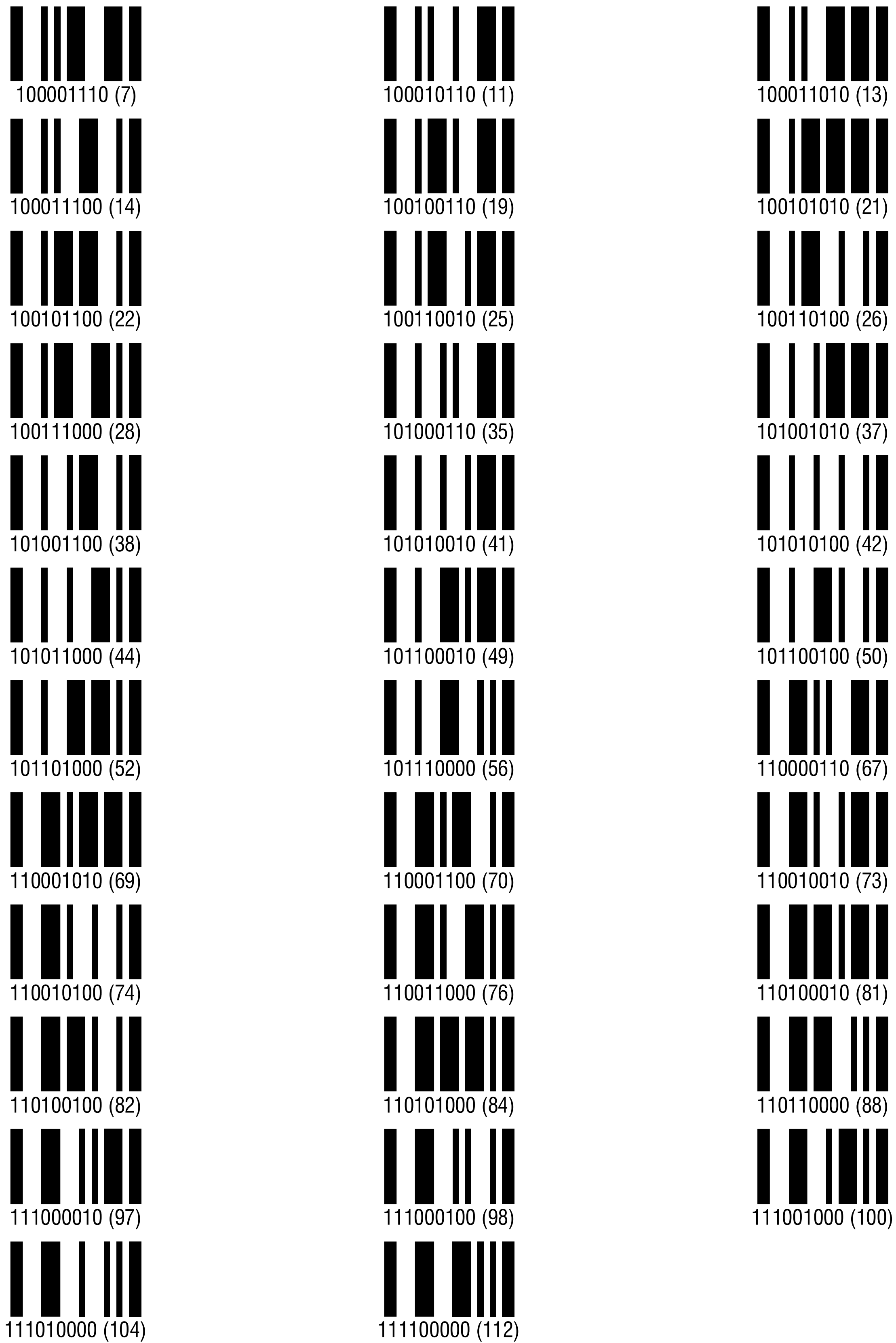
```
27      OWOUT owio, 0, ["<", FNDNXT|FWD|DRKEDG, FNDNXT|FWD|BRTEDG,
28                           FNDNXT|FWD|DRKEDG, FNDNXT|FWD|BRTEDG,
29                           FNDNXT|FWD|DRKEDG, FNDNXT|FWD|BRTEDG,
30                           FNDNXT|FWD|DRKEDG, FNDNXT|FWD|BRTEDG, ">"]
31      GOSUB Ready
32      OWOUT owio, 0, [DUMPADR, RESULTS]
33      FOR cntb = 1 TO 8
34        OWIN owio, 0, [edga]
35        IF (edga - edgb > cnta) THEN code = code | 1
36        edgb = edga
37        code = code << 1
38      NEXT
39      IF (edgc - edgb > cnta) THEN
40        code = code | 1
41        code = code REV 9
42      ENDIF
43      IF (code & 1 = 0 AND code & 256 AND code <> pcode) THEN
44        cntb = 0
45        FOR cnta = 1 TO 7
46          IF (code & (1 << cnta)) THEN cntb = cntb + 1
47        NEXT
48        IF (cntb = 3) THEN
49          pcode = code
50          DEBUG BIN9 code, BELL, CR
51        ENDIF
52      ENDIF
53    ENDIF
54  LOOP
```

Here's the explanation:

Line	Description
1-10	Some variables that aren't included in the template are defined here.
12	Initial exposure time is set to 30.
13	Set exposure time; binary acquisition parameters: FIXED threshold = 128, Hysteresis = 5; and LED brightness = 127 (50% brightness for duration of exposure).
15	Beginning of the main program loop.
16-18	Queue up and execute as a block: Binary acquisition, count dark edges and bright edges, find the first and last pair of edges. Note: The reset at the beginning terminates the reads at the end of the **DO** loop.
19	Wait for everything that's been queued up to finish.
20	Start dumping memory at **MAXPIX**, the value of the brightest pixel.
21	Read the maximum brightness and the results of the counts and finds. **cnta** gets read multiple times. The first two are just dummy reads to skip over **MAXLOC** and **AVGPIX**.
22-23	Adjust the exposure time, based on the value of **maxbrt**.
24	Check to see if the edge counts are both six and that the widths of the first and last bars are roughly equal.
25	Set the width threshold for wide vs. narrow bars/spaces to the average of the first and last bar widths.
26	Initialize the **code** variable to zero.
27-30	Queue up and execute as a block: find all the remaining edges.
31	Wait until all the buffered commands have executed.
32	Start dumping the edge locations.
33	There are eight edges to read.
34	Read the next edge location.
35	Width of the bar is new edge location minus prior edge location. If it's greater than the threshold width, it's a **1**, so add one to **code**.

36	Save current edge location in previous location.
37	Shift **code** to the left by one.
38	Repeat for the other edges.
39	The last bit is the white bar separating the end bar from the prior edge. Check to see if it's a wide one.
40	It's a wide one, so stick in a **1** bit.
41	This also means we've read the barcode upside down, perhaps, so reverse the bits.
43	Check to see if the last bit is now a **0**, the first bit a **1**, and if this is a new barcode, rather than still hovering over the one we just read.
44-47	It is, so count the wide bars from bits one to seven.
48	Check to make sure that that count is three.
49	If it is, save the new **code** in **pcode.**
50	Send the results in binary to the DEBUG screen and beep the beeper – just like a real barcode reader!
54	Go back and do it all over...

When you run this program, you can hold the motherboard/imager/illuminator stack by hand over the barcode(s) being scanned. The LEDs will be strobing continuously – albeit annoyingly. Whenever a new, properly-formed barcode is encountered, its value will be displayed in the DEBUG screen, and the PC will beep. You may need to vary the subject distance as you scan to get good reads. However, you should virtually never get an erroneous read. There are just too many tests for validity for that to happen with more than miniscule probability.



All 35 barcode combinations (with decimal values that exclude the first and last bits).