Column #106 February 2004 by Jon Williams:

Makin' It Motorized

I don't know about you, but I'm still exhausted by last month's column – wow, that was a real workout, wasn't it? I do hope you found it useful though. This month, we're going to go a lot easier, but still have a bunch of fun! And after its terrific comeback the last couple of months, we're going to have that fun with the venerable BASIC Stamp 1.

There's no denying that personal robotics is one of the fastest growing aspects of hobby electronics. There are clubs all across the globe devoted to it, and each week seems to bring another television show with robotics as its centerpiece. An easy way of getting started in homegrown robotics is to convert an existing motorized toy. They're all over the place, in fact, your family may have a few left-over from the holidays that no longer seem interesting. Why not spice up an old toy with a brain you can program yourself? Okay, then, let's do it.

Micro Motor Control

Since virtually all motorized toys use small DC motors, and controlling them requires full-time PWM that we can't do natively with the Stamp, we'll use an external motor controller. Pololu Corporation makes and sells components devoted to small robotics and their Micro Dual Serial Motor Controller is perfect for our task of converting a motorized toy.

The SMC02 accepts instructions through a serial connection and will control two motors (speed and direction). Motor voltage can be from 1.8 to 9 volts, with currents up to 1 amp per motor! This little dude rocks. And little is accurate; you can see what it looks like in Figure 106.1 next to a penny and a BS1-IC module.

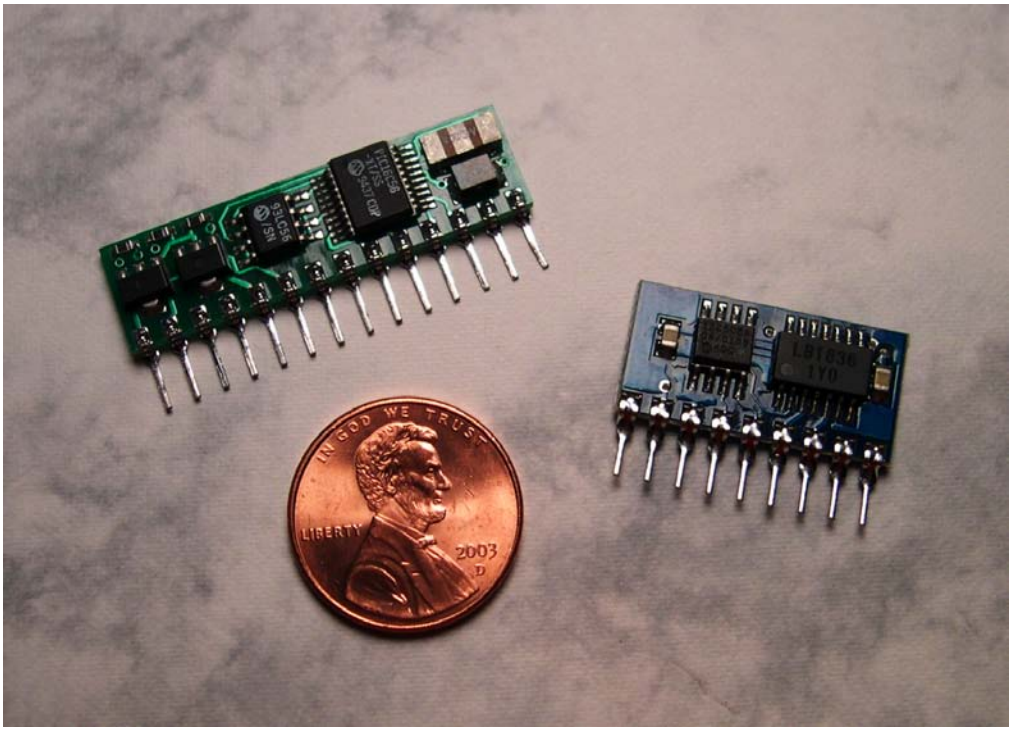

Figure 106.1: The Pololu Motor Controller (right) with a BS1 module

Let's get right into it. A typical first robotic project is a "bumper bot" – the kind of robot that when it bumps into an obstacle will turn away and then keep moving. There are BS2 examples of this kind of robot everywhere; I wanted to see if there was enough space in the BS1 to pull it off. As it turns out there is with room to spare, which means we can add more "intelligence" to our robot once we get in going.

Figure 106.2 shows the schematic for our simple robot controller. Two pins are used to communicate with the Pololu controller (SMC02), two others for our bumper inputs (using our standard "safe" circuit). The first control line to the SMC02 is the serial connection. The SMC02 will automatically detect baud rates from 1200 to 19.2 kBaud. As our top-end limit on the BS1 is 2400, that's what we'll use. The second line controls the Reset input to the SMC02. Using the hard reset line will let us stop both motors at once when required without having to send serial commands.

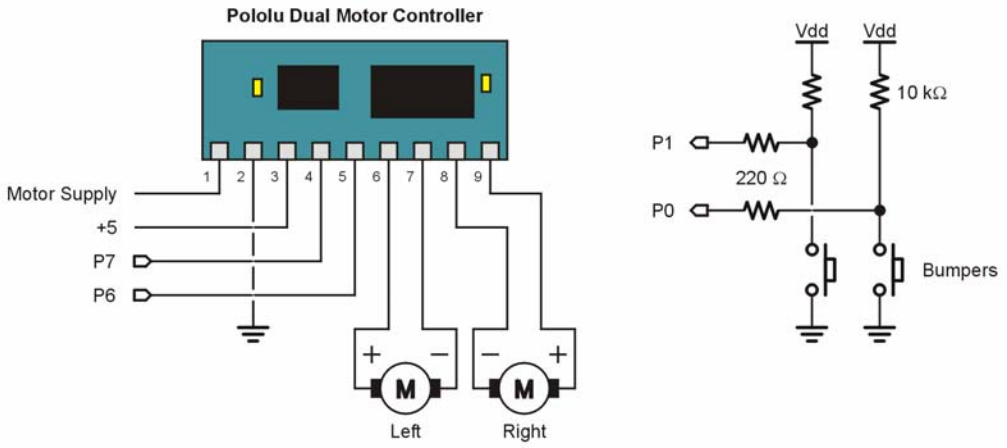

Figure 106.2: Bumper-Bot Schematic

While interfacing to the SMC02 is simple and straightforward, it is very specific. Let's have a look at the setup portion of the program:

```
Setup:
  HIGH SOut
  GOSUB Reset_SMC

...

Reset_SMC:
  LOW Reset
  PAUSE 1
  HIGH Reset
  RETURN
```

The first line sets the idle state of the serial connection. Since we're using a "true" mode this is important so that the start bit (high-to-low transition) of the first transmission is properly detected. Next, the controller is reset by taking the Reset input low and holding there for a millisecond before retuning it high. The Pololu docs suggest that 1 millisecond is overkill, but I found that at least 1 millisecond was required for proper operation.

Okay, the robot is now ready to run. Since our only inputs are the bumpers we'll use them as an indicator to start running the program.

```
Wait_For_Start:
  GOSUB Get_Bumpers
  IF bumpers = %11 THEN Wait_For_Start
  PAUSE 1000

...

Get_Bumpers:
  bumpers = PINS & %00000011
  RETURN
```

We start by scanning the bumpers – a very simple task. You may wonder why I would devote a subroutine call to what is, essentially, as single line of code. The answer is optimism. Huh? Well, I'm pretty sure I'm going to do other things with this core code, so I can simply update this section using different sensors without upsetting the rest of the program. Okay, to collect the state of the bumpers we will read the Stamp input pins and mask those that aren't used (Bit2 – Bit7).

Since our inputs are active-low, we'll get a value of %11 in bumpers when there is no contact. If this is detected at the start, we will loop back to Wait_For_Start until one or both inputs change. Once a bumper input is detected, the program delays for one second to let us get our hand out of the way.

And now we get to the heart of the program. Before we do, let's look at the serial communication between the Stamp and the Pololu Motor Controller. Communication takes place in four-byte packets like this:

```
SEROUT SOut, Baud, ($80, $00, control, speed)
```

The first byte of the packet is a sync byte and will always be $80. The second byte identifies the device type and will be $00 for motor controllers. The third byte, control, identifies the motor number and direction (more in a second), and finally, the fourth byte holds the speed (0 – 127).

The control byte holds the motor direction in bit 0 (1 = forward, 0 = reverse) and the motor number affected by the command in bits 1 – 6 (see Figure 106.3). It may seem odd that there are six bits (0 – 63) for the motor number when there are only connections for two. If you need to control more than two motors you can get additional modules that will respond to different motor number sets. The standard unit has a "-1" at the end of the part number. Addition units are labeled "-2", "-3" and so on.

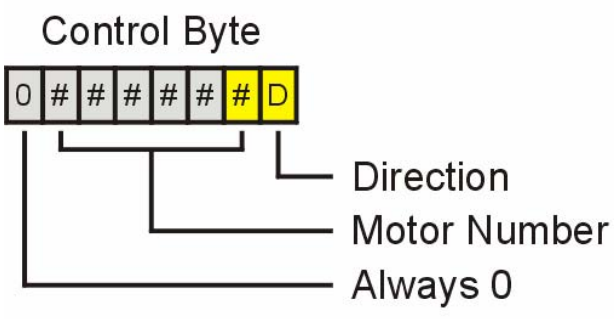

Fig 106.3: Control Byte Mapping

To simplify our program we can create a set of constants for each motor (left and right) and for each direction.

```
SYMBOL  MLFwd   = %11
SYMBOL  MLRev   = %10
SYMBOL  MRFwd   = %00
SYMBOL  MRRev   = %01
```

The reason for the apparent contradiction of the direction bit for the right motor is that the motors installed in a robot are flipped 180 degrees of each other. In order to make the robot go in one direction, the motors must spin opposite each other. If both motors were going forward or reverse, the robot would just spin in place (we use this fact for turning).

Okay, then, the rest of the code should be easy to understand and it's fairly modular so we'll go through it a section at a time.

```
Main:
  GOSUB Get_Bumpers
  IF bumpers = %11 THEN Accelerate
  GOSUB Reset_SMC
  speed = SpdMin
  PAUSE 10
  BRANCH bumpers, (Back_Out, Right, Left)
  GOTO Main
```

The purpose of the main loop is to scan the sensors and **BRANCH** to the code section that handles robot direction control. The first check is for a bumpers value of %11 which means no sensor contact. When there is no sensor contact we will continue forward and accelerate if not already at top speed. When a bumper input is detected, the SMC02 will be reset to stop both motors, the robot speed will be reset to its minimum and **BRANCH** is used to select the object avoidance code.

```
Accelerate:
  speed = speed + SpdRamp MAX SpdMax
  SEROUT SOut, Baud, ($80, 0, MLFwd, speed)
  SEROUT SOut, Baud, ($80, 0, MRFwd, speed)
  PAUSE 50
  GOTO Main
```

When the robot isn't bumping into things it will be allowed to accelerate. The speed is increased up to the SpdMax constant by the value set in SpdRamp. We can control the acceleration by modifying SpdRamp, the delay time, or both.

Our biggest chunk of work will come when we run head on into an object; this is indicated by activity on both bumpers. What we'll want to do is back up, turn out, and then start again.

```
Back_Out:
  SEROUT SOut, Baud, ($80, 0, MLRev, speed)
  SEROUT SOut, Baud, ($80, 0, MRRev, speed)
  PAUSE 250
  GOSUB Reset_SMC
  SEROUT SOut, Baud, ($80, 0, MLFwd, speed)
  SEROUT SOut, Baud, ($80, 0, MRRev, speed)
  PAUSE 500
  SEROUT SOut, Baud, ($80, 0, MRFwd, speed)
  GOTO Main
```

This simple program uses a fixed turn time which will, of course, require adjustment for each robot's specific mechanics (wheel size). A bit of interest can be added by randomizing this delay beyond a specified minimum; this will make the robot look slightly more organic in its behavior.

Turning left or right based on a single bump input is very easy. The motors already been stopped, we simply have to reverse the motor opposite the active sensor, and then proceed forward again.

```
Right:
  SEROUT SOut, Baud, ($80, 0, MRRev, speed)
  PAUSE 200
  GOTO Main

Left:
  SEROUT SOut, Baud, ($80, 0, MLRev, speed)
  PAUSE 200
  GOTO Main
```

Once again, the size of the wheels on our robot will affect how quickly it spins so we may need to adjust the delay timing. Remember that we can "tune" the robot's behavior by adjusting movement delays and speed values. One note is that – depending on the size and weight of your robot – you may need to adjust the SpdMin constant. Some robots will need just a little more *oomph* to get started.

And there you have it; a simple "bumper bot" using the BS1 – and only about half the code space at that. What this means is that we have ample room to add different "behaviors" to the robot. Let's take a look at one such behavior.

Go To The Light....

In nature there are animals that seek light (like moths to a flame). We can mimic this behavior in a robot, and when we do we call the robot a photovore. As our eyes detect light, we can use a photocell to detect light levels for the robot. By using two sensors, we can compare the light levels of each and determine the direction the robot should move to seek the light.

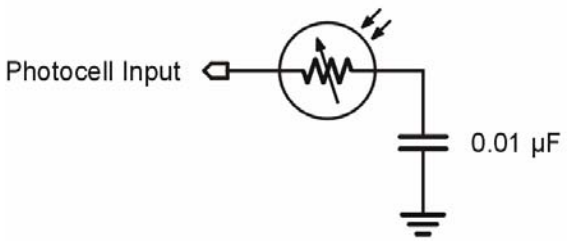

Fig 106.4: Photocell Circuit for the BS1 POT Function

Figure 106.4 shows the connection of a single photocell to the BS1 that will be read with the **POT** function (we'll need two of these circuits for our robot "eyes"). Note that the configuration of the RC components is different that those used with the BS2's **RCTIME** function. The reason for this is that **POT** actually does more than **RCTIME**. The **POT** function charges the capacitor by taking the specified pin high for 10 milliseconds. It then starts a timer and *actively* discharges the capacitor by taking the pin low momentarily, then changing it to an input and checking to see if the capacitor voltage has dropped below the pin's switching threshold. Scott Edwards wrote a very detailed description of **POT** versus **RCTIME** in back in Column #15 (May 1996 – If you don't have the issue, you can download the column as a PDF from the Parallax web site).

In order to experiment with photovore code I created a test program. I will leave it to you to fold it into your robot. The reason I created an experiment program to share here is that I want to encourage you to do the same thing with your projects. Yes, I am a very big proponent of knowing the outcome of a project (having a specification), but we don't have to get there in one step. I frequently get requests for assistance from youngsters who are just starting out and try to solve everything at once. I'm like a broken record and will continue to suggest that we work with intermediate or experimental programs before "going for the gold."

Okay, so what do we want to do? Knowing that the more light on the photocell will cause its reading to fall, we will read both sensors and move in the direction of the smallest reading. That's the broad stroke. What we'll find, however, is if our logic remains that simple the robot will be too sensitive and will appear "twitchy." We solve the twitch by adding a threshold to the readings; that is, the difference between the two readings must exceed a certain threshold before we actually move. This helps accommodate differences in components and smoothes things out.

Finally – and I only learned this after running the program a while – we need to accommodate the sensors being subjected to conditions where the readings will fall to zero. Due to the component values used, this can happen in total darkness or when the sensor is bombarded by bright light (swamped).

Before we get into the code we need to run a calibration program to determine the scaling factor required by the **POT** command. Like **RCTIME**, **POT** reads a 16-bit value but it will scale it to 8-bits for the program. The idea is to set the scaling factor such that the maximum value of the **POT** function is 255.

After connecting our **POT** circuit, we'll select *Pot Scaling* from the editor's *Run* menu (Note: This feature became available in Version 2.1, Beta 1). Figure 106.5 shows the dialog for POT Scaling. We must first select the POT pin from a drop-down. Then when we click Start, a short program will be downloaded to the Stamp (yes, this overwrites the current program). Normally, we would turn the pot until we get the lowest reading in scaling factor. Since we're using a photocell, we adjust its value by shading it from light. I decided to use a translucent foam cup over my components to determine the scale level (max resistance).

I applied the same scale factor to both circuits and found that I got significantly different values given the same light level (both covered with the foam cup). To adjust for component differences, I tweaked the scaling factor of the second "eye" circuit until both sensors returned the same value under the same lighting.

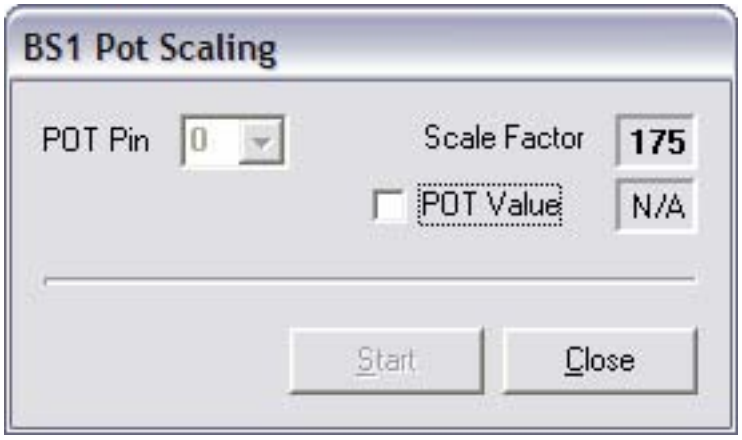

Fig 106.5: POT Scaling Dialog in the Windows Editor

Okay, let's look at the code:

```
Read_Eyes:
  POT EyeL, 145, lfEye
  POT EyeR, 175, rtEye
  DEBUG CLS, lfEye, CR, rtEye, CR

Check_Eyes:
  IF lfEye = 0 AND rtEye = 0 THEN Is_Dark
  IF rtEye < lfEye THEN Is_Right

Is_Left:
  move = %10
  IF lfEye = 0 THEN Eyes_Done
  lfEye = rtEye - lfEye
  IF lfEye >= Threshold THEN Eyes_Done
  move = %00
  GOTO Eyes_Done
```

```
Is_Right:
  move = %01
  IF rtEye = 0 THEN Eyes_Done
  rtEye = lfEye - rtEye
  IF rtEye >= Threshold THEN Eyes_Done
  move = %00
  GOTO Eyes_Done

Is_Dark:
  move = %11

Eyes_Done:
  RETURN
```

The code starts by reading each "eye" into its own variable and displaying the readings in the **DEBUG** window. Then we check to see if both values are zero; this will happen in total darkness (due to **POT** function timeout) or when both photocells are swamped with light. In either case we want the robot to hold still. Setting the move variable to %11 tells our main code that this is the case.

Under most conditions we'll have at least one photocell reading, so we do a comparison to see which side is getting the most light (it will have the lowest value). Let's assume that the left photocell had a lower reading and go through that section of code. The code for the right side works identically.

We'll assume that the threshold is going to be exceeded and preset the move variable to left (%10). Next, we check the value of the left photocell. If it's zero (swamped) there is no need to check the threshold and we can exit. If it is not swamped we will subtract the right photocell reading from the left and compare the result against the threshold. If the threshold is exceeded we exit, otherwise the move variable is set to forward (%00).

Since this is a demo program, we should have some code to show us what's happening with the sensors. Here's a short bit of code to do that:

```
Main:
  GOSUB Read_Eyes
  BRANCH move, (Go_Straight, Go_Right, Go_Left, Stay_Still)

Go_Straight:
  DEBUG "Straight"
  GOTO Main
```

```
Go_Right:
  DEBUG "Right"
  GOTO Main

Go_Left:
  DEBUG "Left"
  GOTO Main

Stay_Still:
  DEBUG "Holding..."
  GOTO Main
```

There's no magic here, we're simply using **BRANCH** to jump to code that will display the direction of robot travel. This lets us test our "robot eyes" on a breadboard. Bend the photocell leads so that they're oriented on the horizontal plane, and then turn them out from each other to expand the field of view. By shining a flashlight spot in front of the photocells the **DEBUG** window will show the robot's movement behavior.

All that's left to do is replace the **DEBUG** statements with appropriate motor controls (and you know how to do that) to create a robotic photovore – kind of like an electronic kitten (cats love to chase flashlight spots), only this one won't scratch furniture or need a litter box!

Well, I told you we'd take it easy this time, and hopefully you agree with me that this was no less fun that we usually have. Modifying an existing motorized toy is a great way to get kids into robotics – no matter what their age.

Until next time, Happy Stamping.

```
' =============================================================================
'
'   File..... Bumper_Bot.BAS
'   Purpose... Simple bounce-off-object robot core
'   Author.... Jon Williams
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 07 DEC 2003
'
'   {$STAMP BS1}
'   {$PBASIC 1.0}
'
' =============================================================================


' -----[ Program Description ]---------------------------------------------
'
' This program uses the BS1 module and a Pololu Micro Motor controller to
' form the heart of a simple robotic platform.  The small size of the pars
' allow it to be "hacked" into a motorized toy for customization.


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


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

SYMBOL  SOut            = 7                     ' serial out to Pololu
SYMBOL  Reset           = 6                     ' Pololu reset control
SYMBOL  BumpL           = PIN1                  ' left bumper input
SYMBOL  BumpR           = PIN0                  ' right bumper input


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

SYMBOL  Baud            = T2400                 ' can be T1200 or T2400

SYMBOL  Hit             = 0                     ' inputs are active-high

SYMBOL  MLFwd           = %11                   ' left motor forward
SYMBOL  MLRev           = %10                   ' left motor reverse
SYMBOL  MRFwd           = %00                   ' right motor forward
SYMBOL  MRRev           = %01                   ' right motor reverse

SYMBOL  SpdMin          = 40                    ' minimum start speed
SYMBOL  SpdMax          = 125                   ' top speed
SYMBOL  SpdRamp         = 5                     ' delta for acceleration
```

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

SYMBOL  ctrl            = B0                    ' motor control byte
SYMBOL  direction       = BIT0                  ' 0 = CW, 1 = CCW
SYMBOL  motor           = BIT1                  ' 0 = right, 1 = left
SYMBOL  speed           = B1                    ' motor speed
SYMBOL  bumpers         = B2                    ' bumper inputs
SYMBOL  state           = B3                    ' sensor state


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


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

Setup:
  HIGH SOut                                     ' idle state of serial line
  GOSUB Reset_SMC


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

Wait_For_Start:
  GOSUB Get_Bumpers                             ' can bumper inputs
  IF bumpers = %11 THEN Wait_For_Start          ' wait until one or both
  PAUSE 1000                                    ' ... hands out!

Main:
  GOSUB Get_Bumpers                             ' scan inputs
  IF bumpers = %11 THEN Accelerate              ' if not hits, speed up
  GOSUB Reset_SMC                               ' contact -- stop motors
  speed = SpdMin                                ' reset speed
  PAUSE 10
  BRANCH bumpers, (Back_Out, Right, Left)
  GOTO Main

Back_Out:
  SEROUT SOut, Baud, ($80, 0, MLRev, speed)     ' back up
  SEROUT SOut, Baud, ($80, 0, MRRev, speed)
  PAUSE 250
  GOSUB Reset_SMC                               ' stop motors
  SEROUT SOut, Baud, ($80, 0, MLFwd, speed)     ' turn (clockwise)
  SEROUT SOut, Baud, ($80, 0, MRRev, speed)
  PAUSE 500
  SEROUT SOut, Baud, ($80, 0, MRFwd, speed)     ' both forward now
  GOTO Main

Right:
  SEROUT SOut, Baud, ($80, 0, MRRev, speed)     ' spin right
  PAUSE 200
  GOTO Main
```

```
Left:
  SEROUT SOut, Baud, ($80, 0, MLRev, speed)     ' spin left
  PAUSE 200
  GOTO Main

Accelerate:
  speed = speed + SpdRamp MAX SpdMax            ' increase speed to max
  SEROUT SOut, Baud, ($80, 0, MLFwd, speed)     ' both motors forward
  SEROUT SOut, Baud, ($80, 0, MRFwd, speed)
  PAUSE 50
  GOTO Main


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

Reset_SMC:                                      ' reset motor controller
  LOW Reset
  PAUSE 1
  HIGH Reset
  RETURN

Get_Bumpers:
  bumpers = PINS & %11                          ' mask non-bumper pins
  RETURN
```

```
' ===================================================================
'
'   File..... Robot_Eyes.BAS
'   Purpose... Demonstrates Photovore logic
'   Author.... Jon Williams
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 07 DEC 2003
'
'   {$STAMP BS1}
'   {$PBASIC 1.0}
'
' ===================================================================


' -----[ Program Description ]---------------------------------------
'
' Demonstrates photovore (light seeking) logic that can be applied to a
' a Stamp controlled robot.


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


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

SYMBOL  EyeR            = 0                     ' right eye RC input
SYMBOL  EyeL            = 1                     ' left eye RC input


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

SYMBOL  Threshold       = 20                    ' difference threshold


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

SYMBOL  rtEye           = B2                    ' right eye value
SYMBOL  lfEye           = B3                    ' left eye value
SYMBOL  move            = B4                    ' calculated move


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


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

Setup:


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

```
Main:
  GOSUB Read_Eyes
  BRANCH move, (Go_Straight, Go_Right, Go_Left, Stay_Still)

Go_Straight:
  DEBUG "Straight"
  GOTO Main

Go_Right:
  DEBUG "Right"
  GOTO Main

Go_Left:
  DEBUG "Left"
  GOTO Main

Stay_Still:
  DEBUG "Holding..."
  GOTO Main


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

Read_Eyes:
  POT EyeL, 145, lfEye                          ' read robot "eyes"
  POT EyeR, 175, rtEye
  DEBUG CLS, lfEye, CR, rtEye, CR               ' display eye values

Check_Eyes:
  IF lfEye = 0 AND rtEye = 0 THEN Is_Dark       ' too dark or eyes swamped
  IF rtEye < lfEye THEN Is_Right                ' look for dominant side

Is_Left:
  move = %10                                    ' indicate left
  IF lfEye = 0 THEN Eyes_Done                   ' eye is saturated
  lfEye = rtEye - lfEye                         ' get difference
  IF lfEye >= Threshold THEN Eyes_Done          ' check threshold
  move = %00                                    ' near center -- straight
  GOTO Eyes_Done

Is_Right:
  move = %01
  IF rtEye = 0 THEN Eyes_Done
  rtEye = lfEye - rtEye
  IF rtEye >= Threshold THEN Eyes_Done
  move = %00
  GOTO Eyes_Done

Is_Dark:
  move = %11
```

```
Eyes_Done:
  RETURN
```