

# EXTENDING THE SCRIBBLER ROBOT



BY PHIL MASS

## *Surface Detection and a Simple Behavior System*

**T**he Scribbler is an exciting new robot from Parallax. This small blue robot is fully programmable and packed with sensors. As one of the developers of the Scribbler, I can assure you that we tried to add as much capability as possible to the robot while still keeping it affordable for the average person. In this article, I'll give you a quick introduction to the Scribbler and then explain a project I created using a simple behavior system and a creative use for the line following sensor.

### **The Scribbler**

The Scribbler is a very capable robot. Its features include two infrared obstacle avoidance sensors, two line following

sensors, three light sensors, a stall sensor, three user-controlled LEDs, a speaker, and two drive motors with both continuous and timed commands. It also has a rich programming environment so that you can use these sensors and actuators in any possible combination.

There are two ways to program the Scribbler: either graphically, using the Program Maker GUI; or with text in the PBASIC language, using the BASIC Stamp Editor. The GUI makes it easy to program the Scribbler, even if you have no prior



**FIGURE 1.** The Scribbler robot on the checkerboard section of the project arena.



**FIGURE 2.** The line follower sensor on the bottom of the Scribbler.

knowledge of programming or robotics. You simply construct your program by assembling virtual blocks together in the work area. Each programming block performs a function, such as making the motors drive a certain speed or checking the obstacle sensor to see if there is something in front of the Scribbler. In no time, you can have a working program to download to your robot. The GUI also provides branching, loops, logic, and subroutines, so that even advanced programmers will find it a useful tool for quickly building complex programs.

The second way to program the Scribbler is by using the popular BASIC Stamp Editor. Since the Scribbler has a BASIC Stamp 2 as its brain, you can use the same programming tools used to program any Parallax Stamp processor. Writing programs directly in PBASIC gives you full control over every aspect of the Scribbler, and will also familiarize you with the details of how all the sensors and actuators function. With it, you can use the Scribbler to carry out your own robotics experiments and ideas. For my project, I chose to use direct PBASIC programming because I wanted to use my own behavioral control system and to utilize the line sensor in a novel way. You can download both programming tools, along with detailed programming tutorials, for free from the Scribbler Robot website ([www.scribblerrobot.com](http://www.scribblerrobot.com)).

## Sensors and Motors

The Scribbler Robot has three light sensors that look out

### WHERE TO BUY A SCRIBBLER

[www.scribblerrobot.com](http://www.scribblerrobot.com)

The Scribbler robot can be purchased directly from Parallax or any of the distributors listed on the official website. You can also learn more about the Scribbler's features, join the user's forum, and download all of the free software and programming guides.

from the front through three holes that resemble eyes (Figure 1). The light sensors detect the light levels in front of the Scribbler and are roughly focused so that each sees a different area. The short plastic ribs on top of the robot just above the holes point in the direction to which each light sensor looks. These sensors can be used to make the Scribbler drive toward or away from any light source, or to align itself to a light. Since the sensors each look in a different direction, they can also be used as a crude three-pixel vision system. They can also be used to simply detect overall light levels, which is how I use them in my project.

In Figure 1, you can also see the Scribbler's two infrared (IR) emitters mounted in cylinders at its front corners. The two emitters and a single IR detector are used to detect obstacles. When you turn on the emitters, they shoot light out in front of the Scribbler. If there is an object in front of the robot, some of this IR light will reflect back to the detector. By turning the emitters on and off in turn, you can also tell if the object is on the right or left side. This sensor uses a modulated light signal, meaning that when the emitters are turned on, they are actually oscillating at a frequency of 38 KHz. If you send out a slightly different frequency, it will reduce the sensitivity of the detector. So, by sweeping through several frequencies, you can detect objects at different distances.

On the bottom of the Scribbler is a pair of line sensors (Figure 2). Like the obstacle detector, the line sensors detect IR light, this time reflected off of the ground. By measuring the amount of IR light that is reflected back, the robot can tell if the sensor is over a white or black surface. You can use this sensor to follow a black line printed or drawn on white paper. In my project, I'll explain how to use this sensor to detect other floor patterns, as well.

The Scribbler also has a stall sensor that detects when it is stuck and its wheels have stopped turning. If the Scribbler hits something when it is driving around, it can sense the obstacle and free itself.



whether the light was on, I decided to add together the values of the three light sensors in order to reduce the variation of what each sensor detected. Next, I assumed that the light would be on when the robot was reset, and so I stored the initial light level at the beginning of the program. From then on, if the sum of the light sensors became greater than 2.5 times the initial value, I concluded that the room light was off. If the sum was less, then the room light was on (smaller light sensor values mean more light).

To determine the floor color, I needed to differentiate

between white, black, and the gray checkerboard pattern. To do this, on each update cycle, I sampled both of the line sensors 50 times quickly. On each sample, and for each of the two sensors, I added one to a sum if the sensor detected black. So, at the end of the sampling, the sum was within the range of 0 to 100. From testing, I found that if the floor was white or black, both sensors always detected those colors. Therefore, a sum of 0 meant white, a sum of 100 meant black, and anything in between was considered gray. I found that this made for a reliable surface color detector.

## EXAMPLE SCRIBBLER BEHAVIORS

This PBASIC code shows the main behavior loop used in the project along with the complete code for the three lowest priority behaviors. Each behavior checks sensor values to decide if it wants control of the Scribbler, and then compares its own priority constant (such as WhiteDriveP) to the current value of the behavior arbitration variable.

```

` Behavior loop
DO
  GOSUB checkSensors
  GOSUB blackTurnBeh
  GOSUB switchBeh
  GOSUB grayTurnBeh
  GOSUB whiteDriveBeh
  GOSUB whiteTurnBeh
  GOSUB grayDriveBeh
  GOSUB arbiter
LOOP

` Behavior subroutines

whiteDriveBeh:
  IF(light_on = 1) THEN
    IF(behavior < WhiteDriveP) THEN
      behavior = WhiteDriveP
      motor_r = DriveVel
      motor_l = DriveVel
    ENDIF
  ENDIF
  RETURN

whiteTurnBeh:
  IF(floor = 0) THEN
    IF(behavior < WhiteTurnP) THEN
      behavior = WhiteTurnP
      motor_r = TurnVelRev
      motor_l = TurnVelFwd
    ENDIF
  ENDIF
  RETURN

grayDriveBeh:
  IF(behavior < GrayDriveP) THEN
    behavior = GrayDriveP
    motor_r = DriveVel
    motor_l = DriveVel
  ENDIF
  RETURN

```

## Behavior Systems

There are a variety of ways to control a mobile robot. I've found that one of the most effective ways is to use a simple version of behavioral control. With this approach, you write a collection of individual pieces of code called behaviors, each of which performs a simple task. Then, you link all of these behaviors together by assigning them all priorities and connecting them through an arbiter. The arbiter gives control of the robot to the highest priority behavior that requests control. For this project, I used a simple arbiter that employs strict behavior priorities. This means that the priority numbers of the behaviors don't change and that no two behaviors have the same priority. A higher priority behavior can always take control from a lower priority behavior.

The behavior system is run in an infinite loop. Each time through the loop, each behavior decides whether it wants control of the robot by looking at the current state of the sensors. The behavior with the highest priority that wants control is selected and its desired commands are sent to the motors for that cycle. Each behavior cycle, a new behavior can be selected, but most behaviors end up controlling the robot for several cycles at a time. For instance, let's say we have a system with four behaviors with priorities one through four. In a specific cycle, let's say behaviors one, two, and four all want control. Behavior four is granted control because it has the highest priority. When it has completed its task several cycles later, it no longer wants control, so control then passes to behavior two.

This simple type of behavioral control can give a robot very complex overall behavior while using very little RAM memory. The separate behaviors also make it easy to design, as you often only need to think about one behavior at a time.

## Behavior System Details

Now, let's get into the details of the behavior system I used for my project. The behavior system is run in an infinite loop that has three parts: sensors, behaviors, and arbiter.

In the first section, the sensors are read and the current sensor values are stored in variables. The behaviors all look at these stored sensor values so that they all have the same information each cycle. For this project, the sensor section

checks whether the room light is on and samples the floor color. In the second section of the loop, all of the behaviors are run and they each decide whether they want control and, if they do, what motor commands they want to send. In the third section, the arbiter takes the winning behavior's commands and calls the motorSet subroutine to actually set the wheel speeds. In my code, each of the behaviors, as well as the sensor update and arbiter, is a separate subroutine. So, the main loop is a series of subroutine calls: first the sensors, then the behaviors, followed by the arbiter.

I use a variable named behavior\_p to control the arbitration. There are a total of six behaviors in the system, with priorities one through six. At the beginning of each cycle, the behavior\_p variable is set to zero. As each behavior is run, if it wants control, it checks to see if its priority is greater than the current value of behavior\_p. If it is, it sets behavior\_p to its priority and also sets the variables motor\_r and motor\_l with its desired wheel speeds. If its priority is lower than the value of behavior\_p, it does nothing. In this way, once all of the behaviors have been run, the one with the highest priority that wants control will have its priority written in the behavior\_p variable and its desired motor commands in the motor speed variables. Thus, all the arbiter needs to do at the end is to call motorSet to actually set the wheel speeds. The arbiter in my system also displays the priority of the behavior that is currently in control in binary using the three green

user LEDs. That way, it's easy to watch the robot and see how the behavior system is acting as the Scribbler drives around the arena.

## Project Behaviors

Now that we have the general framework, what specific behaviors are needed to implement the project? I've built them up from the bottom, with the lowest priority behavior first.

With the lights off, we want the robot to drive straight on gray surfaces and turn when it reaches white or black surfaces. For our lowest priority behavior, we'll make a behavior called grayDrive that always drives straight. So, if no other behaviors want control, the robot will just drive straight. This is the right behavior if it is driving on a gray surface, but if it drives over a white surface, it should turn, so we need to add another higher priority behavior named whiteTurn to do that. If the robot sees a white surface, the behavior whiteTurn will take control and turn the robot. If it doesn't see a white surface, grayDrive will take control and it will drive straight. This is a good start, but we also want the Scribbler to turn if it sees a black surface. Therefore, we'll add an even higher priority behavior that turns when it sees black called blackTurn. Now, with these three behaviors, the robot acts the way it should with the lights off. It drives straight on gray and turns

if it is on white or black. To recap, our behavior system has three behaviors:

blackTurn — highest priority  
whiteTurn  
grayDrive — lowest priority

That behavior is correct when the lights are off, but when the lights are turned on, we want the robot to drive straight on white and turn on gray or black, so we'll need to add some more behaviors. First, we want the robot to drive straight on white when the lights are on. So, we'll add a behavior called whiteDrive which checks that the lights are on and drives straight if they are. We'll need to give this behavior higher priority than whiteTurn in order to override the turning behavior on white when the lights are off. We also need a behavior that will turn the robot when the lights are on and it sees a gray surface. This behavior, in turn, needs to be higher priority than whiteTurn. Whether the light is on or off, we want the robot to always turn when it sees black. Since we already have a behavior that does this, we don't need to add another behavior, but we need to leave the blackTurn behavior as the highest priority behavior so that, no matter what, the robot will always turn when it sees a black surface. Now, our system has five behaviors:

blackTurn — highest priority  
grayTurn  
whiteDrive  
whiteTurn  
grayDrive — lowest priority

The system works as planned, except there is still one problem. Imagine that the lights are off and the robot is driving straight on a gray surface with the grayDrive behavior. Now, if you turn on the lights, the grayTurn behavior will take over and the robot will start turning. But because it is in the middle of a gray surface, it will just keep turning in place, stranded on the gray surface. This isn't what we want. We want the robot to drive over to a white surface before the lights-on behaviors take control. Therefore, we need one last behavior to help with the transition between the lights being on and off.

We'll name this behavior switch and make it higher priority than all of the gray and white behaviors. If the lights change, this behavior will take control and drive the robot straight until it finds the right floor color. If the lights are

turned off, it drives until it sees a gray surface. If the lights are turned on, it drives until it sees a white surface. This way the robot will never get stuck turning in place. This behavior is still lower priority than the blackTurn behavior, though, so that the robot doesn't drive off the end of the arena. So, our final behavior system has six behaviors:

blackTurn — priority 6  
switch — priority 5  
grayTurn — priority 4  
whiteDrive — priority 3  
whiteTurn — priority 2  
grayDrive — priority 1

## Testing

With all of the behaviors in place, it was time to test the system to make sure it worked as planned. When I was testing, I noticed that there was a problem with the switching behavior. The problem stemmed from the surface color detector, which sometimes gave the wrong answer. When the robot drove from a white surface to a black surface, it would temporarily think that the surface was gray. This would sometimes turn the switch behavior off prematurely and strand the Scribbler in the wrong place. The switch behavior was turning off too easily.

To solve this, I added a variable called off\_cnt and every time the Scribbler saw the right floor surface, it incremented the variable. Only when the switch behavior has seen the right floor surface eight cycles in a row does it turn off and let the lower priority behaviors take control. I also added a beeping sound when the robot was switching so that it was easy to tell what was going on when the robot was transitioning.

## Future Expansion

There are many ways to expand on this project. Instead of just detecting black, white, and a 50% checkerboard, you could expand the sensor to detect other values between black and white, such as surfaces that are 75% black or 75% white. You could also try checkered patterns with smaller squares to speed up the detection time or use non-square patterns such as a field of dots.

Floor color detection could also enable all kinds of robot sports or other competitions with the Scribbler. For instance, if you made a robot soccer field with different patterns on different parts of the field, the Scribbler could detect when to play offense or when to play defense. In other competitions, the Scribbler could detect when it was on its own territory or its competitor's territory. It could also use surface patterns to identify special areas, such as its home base.

In this project, I tried to push the Scribbler into some new territory with a behavior system and a surface detector. With all of the possibilities out there, I'm excited to watch what other people create with their own Scribbles. **SV**

## ABOUT THE AUTHOR

Phil Mass develops robots and other electronic products at Element Products, Inc. ([www.elementinc.com](http://www.elementinc.com)), where he helped to design the Scribbler robot. Previously, he wrote the complete operating software for the original Roomba robotic vacuum cleaner while at iRobot Corp. He can be reached at [pmass@elementinc.com](mailto:pmass@elementinc.com)