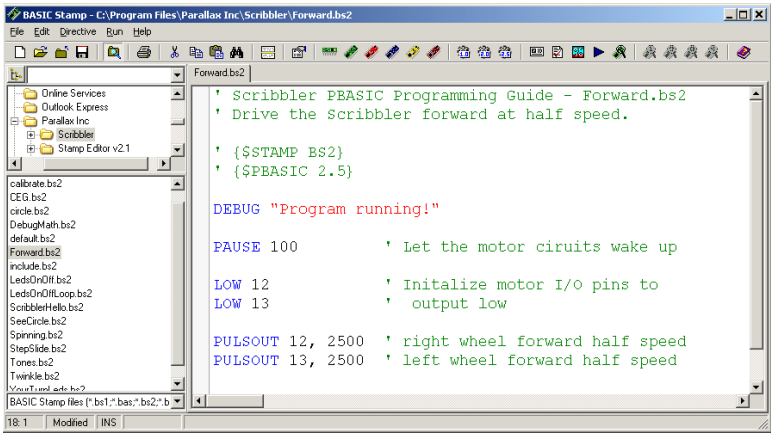
Scribbler PBASIC Programming Guide

Writing Programs

Inside the Scribbler Robot is a small computer called a BASIC Stamp® microcontroller. It performs a list of instructions that make the Scribbler operate. With the BASIC Stamp Editor, you can write your own list of instructions, called a program, and load it into the BASIC Stamp inside your Scribbler Robot. These programs are written in the PBASIC programming language.

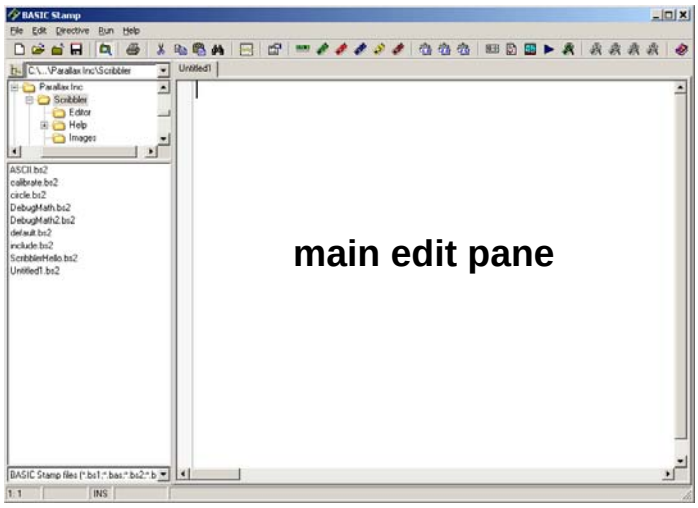


This Guide will show you how to write your first PBASIC programs for your Scribbler Robot. You will be able to communicate with the BASIC Stamp, and make the Scribbler blink its lights, generate sounds, and drive its wheel motors. With the obstacle and stall sensors, you can program the Scribbler to drive safely.

To program your Scribbler Robot, you need to have the BASIC Stamp Editor software (v2.1 or higher) installed and running on your personal computer. You will need to have your Scribbler Robot connected to you computer with the serial cable. Also, you will need to have confirmed that your computer is communicating with the BASIC Stamp microcontroller inside the Scribbler Robot. If you need instructions to do these things, follow the Setting Up Guide before you begin here.

Introducing the BASIC Stamp Editor

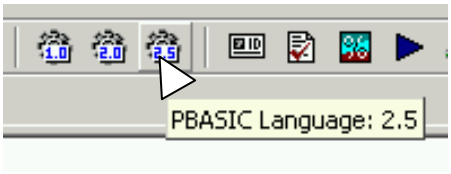
To make a program for your Scribbler, you will type a list of instructions in the **main edit pane**. The programs are written in PBASIC, a language that is easy for people to read and understand. The BASIC Stamp Editor translates the PBASIC program into **binary numbers** - a long string of ones and zeroes - that computers can understand. When you run your program, the Editor transmits it from your computer to the BASIC Stamp inside your Scribbler Robot.



Writing a Program

In every PBASIC program, you must include two instructions that let the BASIC Stamp Editor know what model of BASIC Stamp microcontroller you are programming, and what version of PBASIC you are using. You can use the toolbar icons to place these two instructions in your program automatically, without having to type them.

√ Begin by clicking on the BS2 icon on the top tool bar, since the Scribbler contains a BASIC Stamp 2. This inserts "`' {$STAMP BS2}`" into your program.



√ Next, click on the PBASIC 2.5 icon, for language we will be using in all our Scribbler programs. This inserts "`' {$PBASIC 2.5}`" into your program.

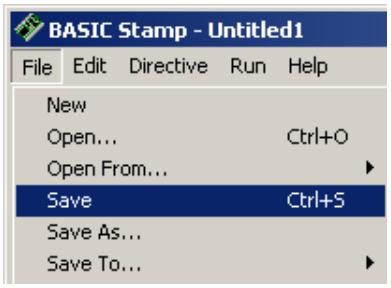
√ Then type in the other 4 lines, so your program looks just like this:

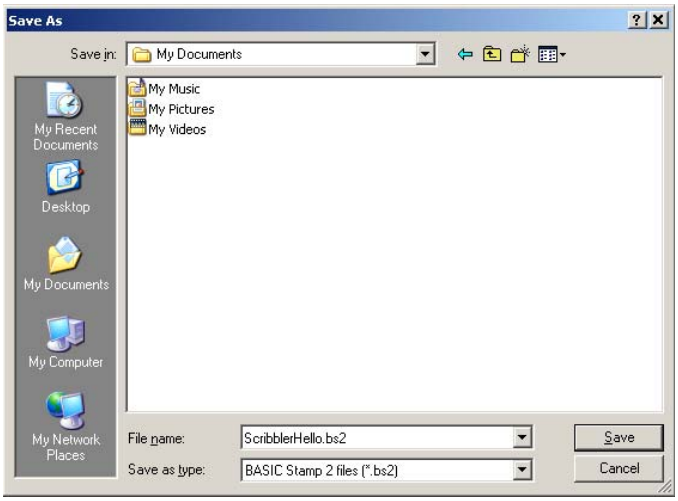
ScribblerHello.bs2

```
' Scribbler PBASIC Programming Guide - ScribblerHello.bs2
' BASIC Stamp sends message to Debug Terminal

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Hello, I am your Scribbler Robot!"
END
```

Now, let's save your program on your personal computer. You don't have to save your program in order to run it, but it is a good habit to save it anyway.

√ From the **File** menu, select **Save**. The Save As window will open.
√ Enter the name ScribblerHello into the **File name** field.
√ Click the **Save** button to save the program and close the window.

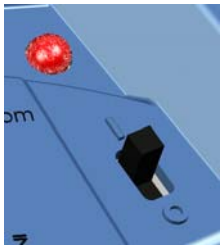




Running a Program

Now your program is ready to be sent to the BASIC Stamp microcontroller inside your Scribbler Robot. First, let's make sure the Scribbler is ready to receive the program.

Programming Connection Checklist



√ Make sure your Scribbler Robot is connected to your computer with the provided serial cable.
√ Make sure the Scribbler power switch is in the "on" position.
1 = ON
0 = OFF
√ Make sure the red power light is on. If it is not, your Scribbler may need new batteries.

If you need instructions to connect your Scribbler Robot to your computer, follow the directions in the Setting Up guide before you continue.

Run the Program

There are several ways to run a program. You can select Run from the Run menu, hold down the Ctrl and R keys together, push the F9 function key, or click the Run icon "▶" on the tool bar.

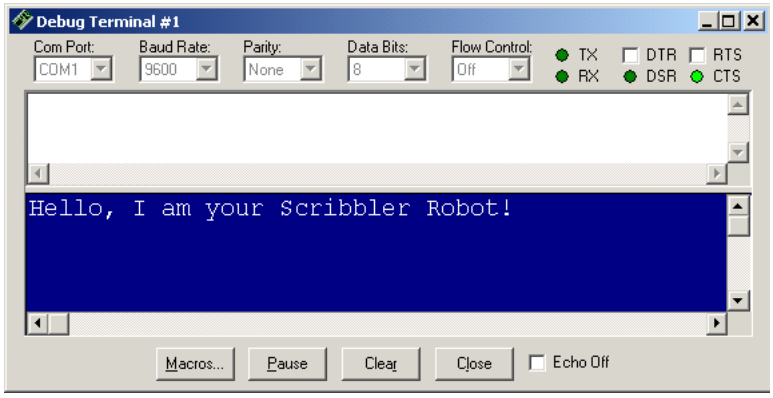
√ Run the program.

You may see a **Download Progress** window open briefly while your program is transmitted from your computer to the BASIC Stamp inside the Scribbler Robot. Or, you may get an error message.

√ If it says "No BASIC Stamps found" recheck your connection and try again.
√ If it says something else, check your program for typing errors and try again.

If all is well, a **Debug Terminal** opens up. The BASIC Stamp transmits the message back to your computer.

√ Press and release the reset button on the Scribbler Robot a few times.



Pressing the reset button makes the program start over from the beginning. So, each time you press it, your message will reprint in the Debug Terminal.

How the ScribblerHello.bs2 Program Works

Let's look at each part of the program to see what it does. The first two lines are **comments**, messages for a person reading the program. It is a good habit to explain your programs with comments so you can remember later what the programs do, and to make it easier for other people to understand them. However, comments are not necessary to make a program run.

Notice that these two lines begin with an apostrophe. This signals to the BASIC Stamp Editor that these lines are comments and do not need to be sent to the BASIC Stamp microcontroller.

```
' Scribbler PBASIC Programming Guide - ScribblerHello.bs2
' BASIC Stamp sends message to Debug Terminal
```

The BASIC Stamp Editor usually ignores everything to the right of an apostrophe, with some exceptions. It does pay attention to special comments called **compiler directives**. These directives are required in every program for your Scribbler Robot.

```
' {$STAMP BS2}
' {$PBASIC 2.5}
```

The first one is the **$STAMP directive**. It lets the BASIC Stamp Editor know that this program is for a BASIC Stamp 2. The **$PBASIC directive** below it indicates the program is written in the PBASIC 2.5 programming language. Remember, it is best to use the toolbar buttons to place compiler directives in your program. If they are mistyped, the program will not run.

The last two lines make up the list of instructions that will be loaded into the BASIC Stamp microcontroller. A **command** is a word that tells the BASIC Stamp to do a certain job. The first instruction in this program uses the **DEBUG** command.

```
DEBUG "Hello, I am your Scribbler Robot!"
```

This command tells the BASIC Stamp to send a message to the personal computer through the serial cable. The text between the quotes will appear in the Debug Terminal.

```
END
```

The second command, **END**, puts the BASIC Stamp into low power mode after the program runs. In low power mode, the BASIC Stamp waits for you to push and release the reset button on the Scribbler Robot, or to load a new program.

More about DEBUG

DEBUG is a powerful command with many uses. Here, it sent a simple text message to your computer, exactly as you typed it. But DEBUG can also be used to perform and report math calculations, prompt the user to perform a task, report the status of a sensor, and much more. We will use DEBUG many different ways throughout this guide. But for now, let's add to our current program to do a little math.

√ From the File menu, choose Save As, and enter the new filename DebugMath.bs2.
√ Change the comment lines to document the new program:

```
' Scribbler PBASIC Programming Guide - DebugMath.bs2
' BASIC Stamp displays math problem answer in Debug Terminal
```

√ Add three more DEBUG commands under the first one:

```
DEBUG CR, "What is 7 x 12?"
DEBUG CR, "The answer is: "
DEBUG DEC 7 * 12
```

Your new program will look like this:

DebugMath.bs2

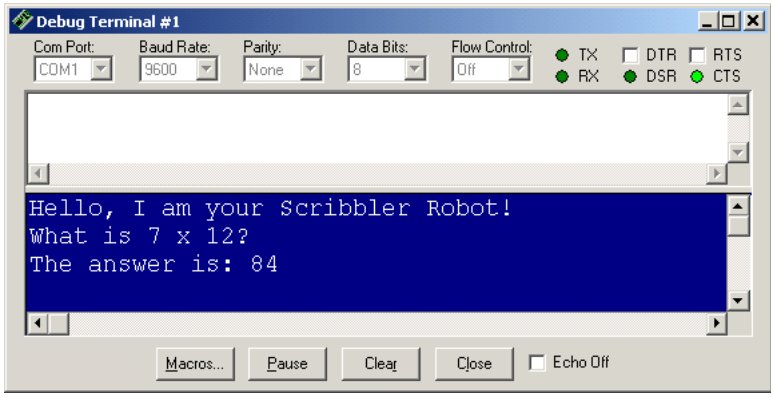
```
' Scribbler PBASIC Programmming Guide - DebugMath.bs2
' BASIC Stamp displays math problem answer in Debug Terminal

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Hello, I am your Scribbler Robot!"
DEBUG CR, "What is 7 x 12?"
DEBUG CR, "The answer is: "
DEBUG DEC 7 * 12
END
```

√ Save the updated program.
√ Run the program.

The Debug Terminal will open, and display the answer to the math problem.

DEBUG Formatters and Control Characters

Notice that this program has four DEBUG commands. Also, notice their messages appear on three separate lines in the Debug Terminal.



In this program, the second and third DEBUG commands are followed by **CR** and a comma. CR stands for "carriage return," and it is an example of a DEBUG **control character**. Control characters move the cursor around in the Debug Terminal, so you can place messages where you want them to appear. By using "**DEBUG CR,**" these messages get printed on their own lines.

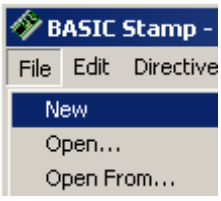
The fourth DEBUG command does not use the CR control character. That is why its message (the answer to the math problem) appears right after the third message on the same line.

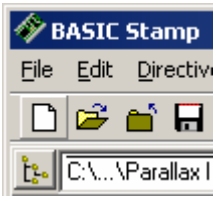
The fourth DEBUG command is followed by **DEC**. This is an example of a DEBUG **formatter**. A formatter determines what form a message will take in the Debug Terminal. The DEC formatter made the answer to the math problem display in the form of a decimal number.

ASCII Code

If you forgot the DEC formatter in this program, your answer would be displayed in its **ASCII code** equivalent, the capital letter "T." ASCII code stands for **American Standard Code for Information Interchange**. Most microcontrollers and personal computers use this code to assign a number to each keyboard function. Some numbers correspond to keyboard actions, such as cursor up, cursor down, space, and delete. Others correspond to letters, numbers, and symbols. The ASCII code numbers 32 through 126 include the characters and symbols that the BASIC Stamp can display in the Debug Terminal. Let's try it.

√ To start a new program in the BASIC Stamp Editor: from the File menu, select **New**. Or, you can click on the toolbar icon that looks like a new sheet of paper.





√ Enter and run the program ASCII.bs2, shown on the next page.

ASCII.bs2

```
' Scribbler PBASIC Programming Guide - ASCII.bs2
' Use ASCII code to in a DEBUG command to display a message.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG 66,65,83,73,67,32,83,116,97,109,112,32,50
END
```

What message did you see in the Debug Terminal? It should print "BASIC Stamp 2."

Your Turn

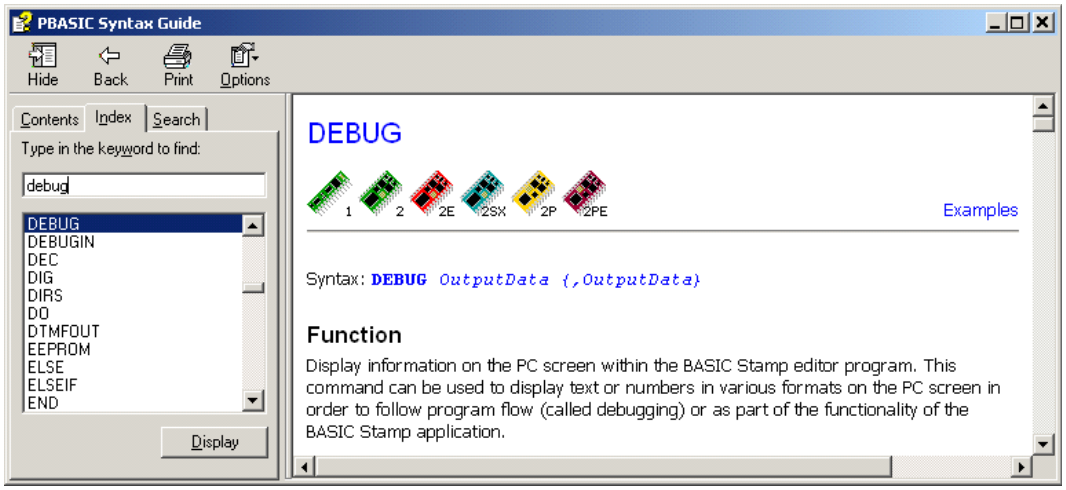
√ Write a program using DEBUG and ASCII code that displays your name in the Debug Terminal. The character codes are included in the ASCII chart at the end of this guide.

Learning More with the Help File

There are many formatters and control characters for the DEBUG command. You can look up more information about DEBUG and every other PBASIC command by using the Help file in the BASIC Stamp Editor. Let's take a look.

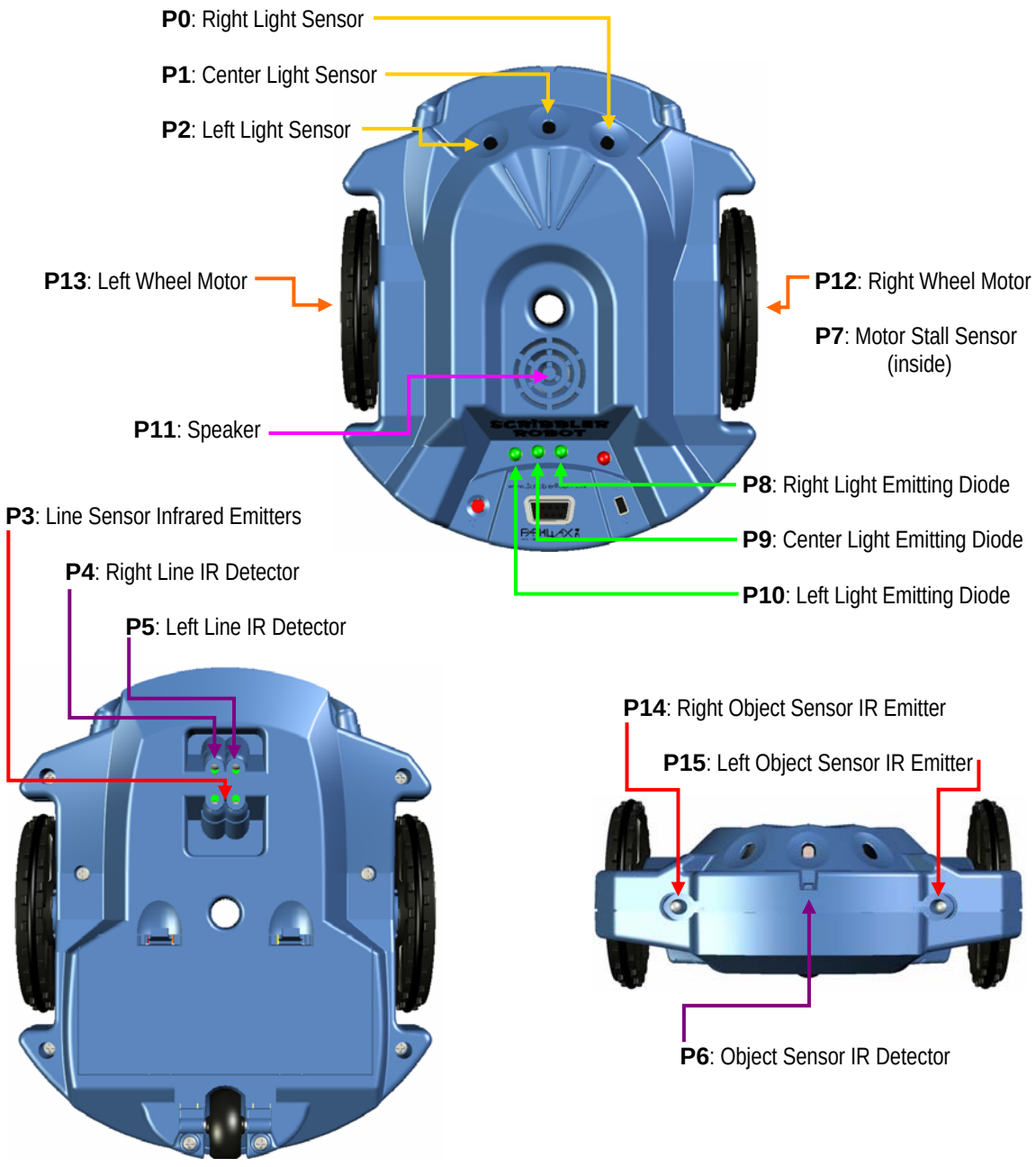
√ From the Help menu choose Index, or click on the Help icon.
√ Under the Index tab, type "debug" into the keyword field.
√ Press the Enter key to open the DEBUG article.

Now you can read all about the DEBUG command, and see charts of all its formatters and control characters. Use the scroll bar to the right of the window to see the whole article.



Scribbler Hardware Programming Connections

So far, our example programs have caused the BASIC Stamp microcontroller to send messages to your personal computer. But the main job of the BASIC Stamp inside the Scribbler is to control the parts of the robot. The BASIC Stamp does this through its 16 **I/O pins**, numbered from **P0** through **P15**. Each I/O pin is connected to one of the Scribbler's circuits.



About the BASIC Stamp I/O Pins

I/O stands for input/output. Each BASIC Stamp I/O pin can do three different things:

- **Connect a circuit to +5 volts**. This makes the pin an **output**, sometimes called "**output high**." In this way, the BASIC Stamp can turn on a circuit in the Scribbler.

- **Connect a circuit to 0 volts (ground)**. This also makes the pin an output, "**output low**." This is how a BASIC Stamp can turn off a circuit in the Scribbler.

  Sometimes a pin is rapidly switched between output high and output low in a specific pattern. In this way, the BASIC Stamp can send a **signal** which other devices can recognize.

- **Monitor the voltage on a circuit.** This is called making the pin an "**input**." In this way, the BASIC Stamp can tell if a circuit is turned on or off, or receive signals from other devices. When an I/O pin is an input, it does not change the voltage of the circuit it is monitoring.

In this Guide, we will use the BASIC Stamp I/O pins to do all these things as we control the Scribbler's circuits. Let's start with a program that blinks the LEDs on and off.

Blinking the Lights with Program Loops

This program will control the Scribbler's three green **light emitting diodes**, called LEDs for short.

√ Enter, save and run the program LedsOnOff.bs2, as it is shown below.

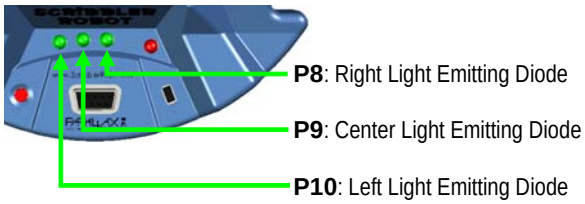
LedsOnOff.bs2

```
' Scribbler PBASIC Programming Guide - LedsOnOff.bs2
' Turn on LEDs for 1 second, then turn them off.

' {$STAMP BS2}
' {$PBASIC 2.0}

HIGH  8
HIGH  9
HIGH  10

PAUSE 1000

LOW 8
LOW 9
LOW 10
```

√ Press and release the Scribbler's reset button a few times.

Each time you re-run the program, the LEDs will turn on for one second, then turn off. Can you see how the program is making this happen?

The **HIGH** command sets a BASIC Stamp I/O pin to output high. So, each of these commands:




```
HIGH  8
HIGH  9
HIGH  10
```

makes an I/O pin connect an LED circuit to 5 volts, which turns on that LED. Even though the BASIC Stamp executes the commands one at a time, it looks like the LEDs all turn on at the same moment. That is because computers work faster than our eyes can see. Sometimes we need to slow them down to human speeds. The next command does just that:

```
PAUSE 1000
```

PAUSE makes the program wait a while before moving on to the next instruction. The number following PAUSE tells the program how long to wait. PAUSE is measured in **milliseconds** (abbreviated **ms**). There are one thousand milliseconds in one second. So, the command PAUSE 1000 makes the program wait for one second before continuing.

```
LOW 8
LOW 9
LOW 10
```

As you might have guessed, the **LOW** command sets a BASIC Stamp I/O pin to output low. This connects P8, P9 and P10 to 0 volts, which turns off each of the LED circuits.

Repeating Actions with DO...LOOP

With this program, we can make the LEDs blink over and over again by pushing the reset button. But what if we don't want to keep pushing the button? We can use a programming concept called a **loop** to make actions repeat automatically.

The DO...LOOP command causes all the instructions between DO and LOOP to be repeated over and over again. In this example, in the first pass through the DO...LOOP the right LED turns on with HIGH 8, and the program pauses for 500 ms. Next, LOW 8 turns the LED off, and the program pauses again for 500 ms. Then, the program reaches LOOP. This signals the program to jump back to DO, and repeat the high-pause-low-pause sequence all over again. The program will continue looping this way endlessly (until the Scribbler is reprogrammed or power is shut off.) This is called an **infinite loop**.

```
DO

   HIGH 8
   PAUSE 500

   LOW 8
   PAUSE 500

LOOP
```

√ Enter, save and run LedLoop.bs2, on the next page.

LedLoop.bs2

```
' Scribbler PBASIC Programming Guide - LedLoop.bs2
' Turn LEDs on and off in a pattern, in an infinite loop.
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

DO
  HIGH  8
  LOW  9
  HIGH  10

  PAUSE 500

  LOW 8
  HIGH 9
  LOW 10

  PAUSE 500
LOOP
```

Now, your LEDs blink on and off in a pattern without having to push the reset button. The DEBUG message “Program running!” should print in the Debug Terminal just once, and not over and over again. This is because the DEBUG command comes before the DO command in the program. It is not inside the loop, so that instruction does not get repeated.

Why is there a DEBUG command in this program?

This command has nothing to do with controlling the LEDs. But, a few personal computers work with the BASIC Stamp better if there is a DEBUG command in every program that uses DO...LOOP or similar commands. Let’s test to see if your computer is one of these.

√ Make the DEBUG command a comment by placing an apostrophe in front of it:

```
'DEBUG "Program Running!"
```

√ Run the program again.

Did your LEDs keep blinking? If they did not, place a DEBUG command in every program as a precaution. The rest of our sample programs include this command as a reminder.

Your Turn

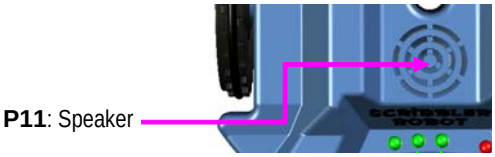
√ Write a program that turns on the LEDs one by one from right to left, 1/10 a second apart. Then, turn them off again one at a time, from right to left. Make the pattern repeat in an infinite loop.

Controlling Sound and Motion with Output Signals

We have used simple HIGH and LOW commands to use the I/O pins to turn circuits on or off. But remember that I/O pins can also send **output signals** by switching between output high and output low in patterns. Output signals are often used to control or communicate with other electronic devices. In the Scribbler Robot, the BASIC Stamp uses special output signals to create sounds on the speaker and to control the wheel motors.

Making Sounds with FREQOUT

The Scribbler Robot's speaker circuit is connected to BASIC Stamp I/O pin P11. There is a special PBASIC command just for making sounds on a speaker: **FREQOUT**. It's short for "frequency out."



The FREQOUT command makes an I/O pin send an output signal in a special pattern. This pattern makes a mechanism inside a speaker vibrate. This vibration causes tiny changes in air pressure that our ears detect as sound. The pace of this vibration is called **frequency**, and it is measured in **hertz** (abbreviated **Hz**). Think about hertz as a measure of vibrations per second. With the FREQOUT command, you can control the frequency of this vibration to create the tone you want. The greater the frequency is, the higher the tone sounds. Here is the **syntax** for FREQOUT:

FREQOUT ***Pin, Duration, Freq1, Freq2***

Pin is the BASIC Stamp I/O pin connected to the speaker
Duration is how long you want your tone to play, in milliseconds, up to 65535.
Freq1 is the frequency in hertz of the tone you want to play.
Freq2 is an optional second frequency, to play two tones at once.

Here is an example FREQOUT command that plays a 1,200 hertz tone for one second:

```
FREQOUT 11, 1000, 1200
```

Pin, Duration, Freq1 and Freq2 are called the **arguments** of the FREQOUT command. Many PBASIC commands have several arguments. The argument values you choose will determine the effect these commands have in the program.

Different speakers respond to different frequencies. The speaker in your Scribbler Robot likes frequencies between 250 Hz and about 2000 Hz. Frequencies in the middle of this range will play the loudest, and those at the ends of this range will play a little quieter.

Here is an example program that plays 5 tones, one at a time. Each tone is generated with its own FREQOUT command

- √ Enter, save and run Tones.bs2

Tones.bs2

```
' Scribbler PBASIC Programming Guide - Tones.bs2
' Play 5 single tones on the speaker.
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program running!"

FREQOUT 11, 200, 500    ' Play 500 Hz tone for 1/5 second
FREQOUT 11, 400, 1000   ' Play 1000 Hz tone for 2/5 second
FREQOUT 11, 600, 1500   ' Play 1500 Hz tone for 3/5 second
FREQOUT 11, 800, 2000   ' Play 2000 Hz tone for 4/5 second
FREQOUT 11, 1000, 2500 ' Play 2500 Hz tone for 1 second
```

Did you hear how each tone played longer and sounded higher than the one before it?

The *Duration* argument is greater with each successive FREQOUT command, so the tone plays longer.

The *Freq1* argument is greater with each successive FREQOUT command, so the tone sounds higher.

```
FREQOUT 11,  200,  500
FREQOUT 11,  400, 1000
FREQOUT 11,  600, 1500
FREQOUT 11,  800, 2000
FREQOUT 11, 1000, 2500
```

Remember, you can mix two tones at once in a single FREQOUT command by using optional *Freq2* argument. You can also make a repetitive sound, like an alarm, by placing FREQOUT commands in a DO...LOOP. This next example program does both:

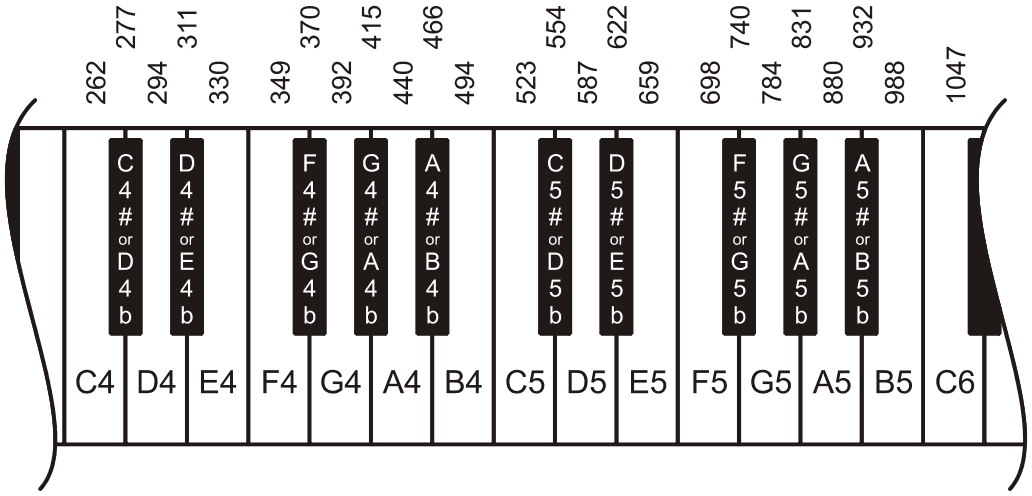
√ Enter, save and run Alarm.bs2, then plug your ears.

Alarm.bs2

```
' Scribbler PBASIC Programming Guide - Alarm.bs2
' Make a alarm noise with FREQOUT commands in a loop
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program running!"

DO

  FREQOUT 11, 200, 700, 750
  FREQOUT 11, 200, 900, 950

LOOP
```

Musical Notes

You can make your Scribbler play more pleasant sounds, like notes similar to those made by piano keys. This picture shows the frequencies for a section of a piano keyboard, starting with Middle C. The frequencies are rounded to the nearest whole number. You can write a song by making a list of FREQOUT commands using these frequencies.



√ Enter, save and run Twinkle.bs2.

Twinkle.bs2

```
' Scribbler PBASIC Programming Guide - Twinkle.bs2
' Play the first 7 notes of Twinkle Twinkle Little Star
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

FREQOUT 11, 500, 523       ' C
FREQOUT 11, 500, 523       ' C
FREQOUT 11, 500, 784       ' G
FREQOUT 11, 500, 784       ' G
FREQOUT 11, 500, 880       ' A
FREQOUT 11, 500, 880       ' A
FREQOUT 11, 500, 784       ' G
```

Your Turn

√ Write a program that plays the first 7 notes of "Mary Had a Little Lamb" or any other song you choose. Hint: If you want to make a note last longer, increase the *Duration*.

Further Investigation

You can learn a lot more about programming BASIC Stamp 2 microcontrollers to play music in the Frequency and Sound Chapter of *What's a Microcontroller?* (v2.0 or higher). This book is included on the Scribbler Software CD, and can be downloaded free from www.parallax.com.

Controlling the Wheel Motors

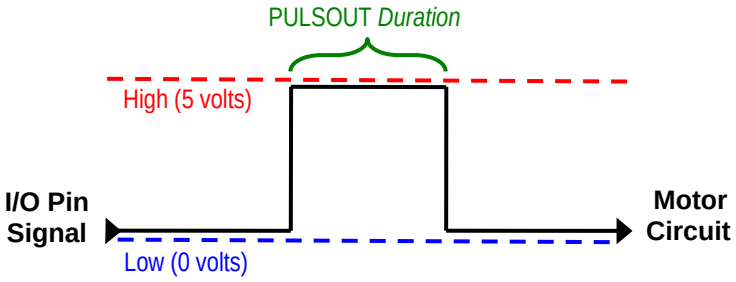
Let's make the Scribbler move! Each drive wheel on the Scribbler Robot has its own motor, which is controlled by its own I/O pin. To make the robot move, you need to send separate commands to each motor. The command that makes the motors move is **PULSOUT**. Just like FREQOUT, PULSOUT causes a BASIC Stamp I/O pin to send a special output signal.

About PULSOUT

PULSOUT sets an I/O pin as an output, then **inverts** its voltage. To invert something is to make it the opposite of what it was – in this case, low voltage becomes high, and high voltage becomes low. The voltage stays inverted for a specified time, then switches back again. The syntax looks like this:

PULSOUT *Pin, Duration*

Pin is the BASIC Stamp I/O pin connected to the motor.
Duration is the amount of time that you want the voltage inverted, measured in **2 microsecond units** (a microsecond is a millionth of a second, abbreviated **µs**).



The Scribbler wheels' motor circuitry is waiting for a high (5 volt) signal pulse, and when it gets one, it will measure how long it lasted.

The length of the high pulse controls the motors **velocity** – the speed and direction combined. Here is how the Scribbler's motor circuitry interprets the high signals:

- The PULSOUT *Duration* argument can be a value from 1000 to 3000
- A *Duration* of 1000 turns a motor full speed backward
- A *Duration* of 2000 makes a motor stop
- A *Duration* of 3000 turns a motor full speed forward
- The closer *Duration* is to 2000, the slower the motor will turn

Initializing the Motors

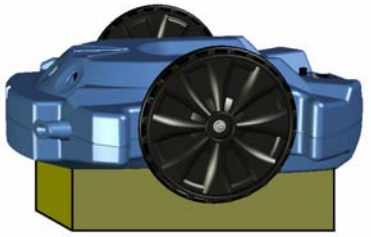
The Scribbler's motor circuitry needs to be **initialized** - prepared to receive a signal. Since the Scribbler's motor circuits are waiting for a high pulse, the I/O pins need to start off sending a low

signal. That way, when the PULSOUT command inverts the signal, it will send a high pulse. So, we need to initialize each motor circuit's I/O pin with a LOW command.

The Scribbler motors' circuitry also needs a little time to start up. If you send them a PULSOUT first thing in a program, the circuits may not be ready to receive the signal. So, we need to place a PAUSE 100 command at the beginning of every Scribbler drive program to give the motor circuitry time to get ready.

Avoiding Accidents

Be careful! When you program the Scribbler to move, it may drive off your desk.



- √ Place your Scribbler Robot on a small book or box, so the wheels don't touch anything.
- √ Enter, save and run FullSpeed.bs2.
- √ Turn of the Scribbler's power switch.
- √ Unplug the serial cable, and set the Scribbler on the floor.
- √ Turn on the power switch, and watch your Scribbler take off. But, be ready to catch it, since it will drive right into things!

FullSpeed.bs2

```
' Scribbler PBASIC Programming Guide - FullSpeed.bs2
' Turn both wheels forward at full speed.
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program running!"

LOW 12              ' Initalize motor I/O pins to
LOW 13              '  output low
PAUSE 100           ' Let the motor ciruits wake up

PULSOUT 12, 3000    ' right motor forward full speed
PULSOUT 13, 3000    ' left motor forward full speed
```

Motor Calibration

This program makes each motor turn forward at its full speed. But each motor is unique, so full speed for one motor might be faster or slower than full speed for the other. So, even though FullSpeed.bs2 uses the same PULSOUT *Duration* of 3000 for each motor, you Scribbler probably won't drive in a straight line.

- √ Place your Scribbler in an open area on the floor, and watch it drive.

Did your Scribbler veer to the left, or maybe to the right? To make it drive straight, you need to reduce the motor speed for the wheel opposite from the way it is veering. Let's replace the PULSOUT *Duration* arguments in FullSpeed.bs2 until your Scribbler drives straight. This is called **software calibration**.

- √ If your Scribbler veered to the left, slow the right wheel with PULSOUT 12, 2900.
- √ If your Scribbler veered to the right, slow the left wheel with PULSOUT 13, 2900.
- √ Run your updated program.
- √ Watch to see if your Scribbler is still veering to the left or to the right.

If your Scribbler is still veering to the same side as before, you need to make that opposite wheel slow down even more. If it is now veering the other way, you can start increasing that PULSOUT *Duration* again, a little at a time.

- √ If your Scribbler is still veering to the same side, keep decreasing the opposite wheel's PULSOUT *Duration* by 100 in each test, until it starts veering the other way.
- √ Once your Scribbler starts veering the other way, start increasing the PULSOUT *Duration* again, by 10 in each test.
- √ If the Scribbler starts veering in the other direction again, start decreasing the PULSOUT *Duration* by 1 until it drives straight, if you are really that patient.

What PULSOUT *Duration* values made your Scribbler drive straight? Write them down here, so you can use them again in your own programs:

```
PULSOUT 12, __________
PULSOUT 13, __________
```

Over time, your Scribbler's motors will become more settled in as the lubricant inside them works its way through the gears. They may begin to perform differently, and you might need to repeat this calibration test in the future.

Basic Maneuvers

Here are some basic maneuvers, how they work, and the PULSOUT *Duration* arguments they use.

Maneuver	Strategy	P12 Duration	P13 Duration
Right Turn	Make right wheel turn slower than left wheel	2500	3000
Left Turn	Make left wheel turn slower than right wheel	3000	2500
Spin Right	Turn wheels in opposite directions	1000	3000
Spin Left	Turn wheels in opposite directions	3000	1000
Back up to the left	Reverse both wheels, left wheel slower	1000	1500
Back up to right	Reverse both wheels, right wheel slower	1500	1000
Full Reverse	Use smallest *Duration* argument for both motors	1000	1000

Your Turn

√ Experiment! Replace the PULSOUT *Duration* values in FullSpeed.bs2 to make the Scribbler perform one of the maneuvers in the table above, or experiment with different *Duration* values to see their effect.

Pin Symbols

You have probably noticed that once you start a wheel turning with a single PULSOUT command, it will keep turning at that velocity until you send it a new command. The BASIC Stamp can make the Scribbler do other things while the wheels are turning. Let's try combining what we have learned so far, in a program that makes the Scribbler's LEDs, speaker, and motors work all together. But before we write a complicated program, let's learn one more helpful trick.

When you write a short program, it is easy to remember which I/O pin number belongs to which circuit. But when you write long programs that control lots of circuits, remembering the pin numbers can be difficult. Also, other people reading the program may not be able to understand what it does.

To make our programs easier to read and write, we can give each I/O pin a name that identifies the circuit it controls. Then, when we write the program, we can use this name, called an **alias**, in place of the I/O pin number. To do this, we use the **PIN** directive at the beginning of a program.

Here is an example that uses the PIN directive to define the I/O pins controlling the LED circuits.

```
LedRight  PIN  8
LedCenter PIN  9
LedLeft   PIN 10
```

Now, when you want to turn the LEDs on or off, you can use the aliases instead of pin numbers:

```
HIGH LedRight
LOW  LedCenter
HIGH LedLeft
```

Example Program: PinSpin.bs2

Let's try a program that uses the Scribbler's LEDS, speaker, and motors together. This program uses pin symbols for each I/O pin needed.

√ Set your Scribbler back on its box so the wheels don't touch anything.
√ Reconnect the serial cable to the programming port.
√ Carefully read the program on the next page, PinSpin.bs2.

Even though it does not have many comments, can you figure out what it will do?

√ Enter, save and run PinSpin.bs2.

PinSpin.bs2

```
' Scribbler PBASIC Programming Guide - PinSpin.bs2
' The Scribbler beeps and blinks green LEDs while spinning
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program running!"

' I/O Pin Definitions

LedRight      PIN 8
LedCenter     PIN 9
LedLeft       PIN 10
Speaker       PIN 11
MotorRight    PIN 12
MotorLeft     PIN 13

' Motor Initialization

LOW MotorRight
LOW MotorLeft
PAUSE 100

' Main Program

PULSOUT MotorRight, 2700
PULSOUT MotorLeft, 1300

DO
  HIGH LedRight
  LOW  LedCenter
  HIGH LedLeft

  FREQOUT Speaker, 250, 523

  LOW  LedRight
  HIGH LedCenter
  LOW  LedLeft

  FREQOUT Speaker, 250, 659
LOOP
```

√ Turn off the Scribbler, disconnect the programming cable, and set it on the floor.
√ Turn the Scribbler back on and watch it perform!

Did the Scribbler do what you thought it would from reading the program?

Making Decisions with Sensors

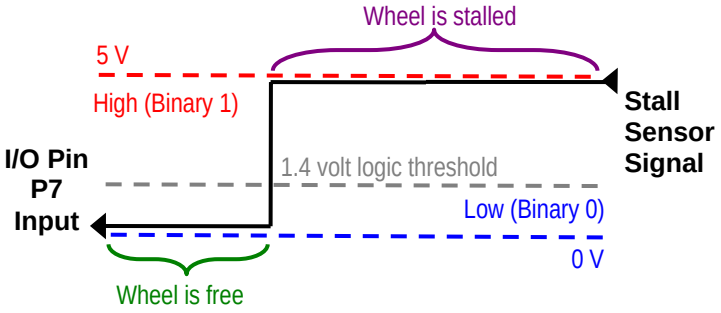
So far, our programs have used a BASIC Stamp I/O pin only as an output: for turning an LED circuit on and off or for sending a control signal to a speaker or motor. Next, we will learn to use an I/O pin as an input, to receive a signal from a sensor. After that, we can use this sensor signal input value to make a decision. This is how the Scribbler can be programmed to respond to its environment independently.

The Stall Sensor

By now, you probably have seen your Scribbler run into an object, then try to keep driving even though it was stuck. The Scribbler has a built-in stall sensor that can tell when the motors are running but the wheels are not turning freely (the Scribbler is stuck). The stall sensor circuit is connected to both motors, and to BASIC Stamp I/O pin P7.

When the motors are running and the wheels can turn freely, the stall sensor sends a low voltage signal to I/O pin P7. If the motors are running but either one of wheels can't turn, the stall sensor sends a high signal to P7.

When an I/O pin is set to input, it can receive this signal by sensing the voltage level on the circuit it connects to. The BASIC Stamp interprets this voltage level using **TTL logic**. This means that any voltage above 1.4 volts is considered a high signal, and voltage below 1.4 volts is considered a low signal. The BASIC Stamp reads a high signal as the value 1 and a low signal as the value 0.



To receive the signal from the stall sensor, I/O pin P7 needs to be set as an input. You can use the INPUT command to do this, but it is often not necessary. Input is the **default mode** for BASIC Stamp I/O pins: they are automatically set to input unless you put a command in a program that uses them as an output.

Defining Variables

So, we know that the BASIC Stamp can interpret the stall sensor signal as a number value: 0 or 1. Now, we need a way for our program to remember that value so it can be used later in other instructions. The most common way to do this is to create a **variable.** A variable is a reserved piece of the BASIC Stamp RAM memory. To create and use a variable, you must **declare** it in your program, usually at the beginning.

Here is the syntax for a variable declaration:

name* VAR *Size

name is the name you choose for your variable. It is best to pick a name that describes what you will use the variable for. There are some rules for variable names:

1. The name cannot be a **reserved word** - one that is already used by PBASIC. Some examples of reserved words you will recognize are DEBUG, HIGH, LOW, and PIN.
2. The name cannot contain a space.
3. The name must begin with a letter or an underscore "_", but it can also include numbers.
4. The name must be less than 33 characters long.

Size is the size of variable that you choose. The BASIC Stamp has 4 sizes of variables: Bit, Nib, Byte or Word. This table shows what range of values can be stored in each type of variable.

Variable Type	Value Range
Bit	0 to 1
Nib	0 to 15
Byte	0 to 255
Word	0 to 65535

It is a good programming habit to use the smallest size variable that you need. For the stall sensor, we only need to see if it is sending a low signal (0) or a high signal (1). So, a Bit will be enough. Here's an example variable declaration we could use for the stall sensor:

```
stuck VAR Bit
```

Example Program: CheckStall.bs2

The next example program will check the status of the stall sensor, and display the status in the Debug Terminal.

- √ Place the Scribbler back on its box so the wheels aren't touching anything.
- √ Reconnect the serial cable to the programming port.
- √ Enter, save and run CheckStall.bs2.
- √ Leave the Scribbler on its box so the wheels can turn freely.
- √ Gently press your hand against one wheel's tire to slow it down, and watch the Debug Terminal.
- √ As soon as you see the number in the Debug Terminal change from a 0 to a 1, release the wheel.
- √ Try it on the other wheel.

When you have applied enough pressure with your hand to keep either wheel from turning freely, the number in the Debug Terminal will change from a 0 to a 1.

CheckStall.bs2

```
' Scribbler PBASIC Programming Guide - CheckStall.bs2
' Display stall sensor I/O pin status in Debug Terminal
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program running!"

' I/O Pin Definitions
Stall      PIN 7
MotorRight PIN 12
MotorLeft  PIN 13

' Variable Declaration
stuck VAR Bit

' Motor Initialization
LOW MotorRight
LOW MotorLeft
PAUSE 100

' Main Program

PULSOUT MotorRight, 3000 ' motors full speed forward
PULSOUT MotorLeft, 3000

DO                       ' Put status of Stall I/O pin
  stuck = Stall          '  into the variable "stuck"
  DEBUG HOME, BIN stuck  ' Display binary value of stuck
  PAUSE 50               ' time to read display
LOOP
```

How CheckStall.bs2 Works

These three lines of code are pin definitions that give names to the I/O pins the program will use.

```
Stall      PIN 7
MotorRight PIN 12
MotorLeft  PIN 13
```

This line declares a bit-sized variable, “stuck,” that will be used to store the status of the stall sensor pin.

```
stuck VAR Bit
```

These three lines should be familiar: they are used in every program that controls the Scribbler's motors. Since we have used the PIN directive, we can use pin names instead of numbers in the LOW commands.

```
LOW MotorRight
LOW MotorLeft
PAUSE 100
```

The first two lines in the main program start both motors turning at full speed forward.

```
PULSOUT MotorRight, 3000
PULSOUT MotorLeft, 3000
```

The rest of the main program monitors the status of the stall sensor's I/O pin, and displays that status in the Debug Terminal. It is nested in a DO...LOOP, so the I/O pin status can be updated continually.

Remember that Stall is the name we declared for I/O pin P7, which is receiving a signal from the stall sensor. Stall will interpret this signal as a value, either a 0 (not stuck) or a 1 (stuck). The instruction causes value of Stall to be stored in the variable stuck – it makes stuck equal to whatever Stall is at that moment.

```
DO
  stuck = Stall
```

The DEBUG command displays the value of the stuck variable in the Debug Terminal. This DEBUG command is using a new control character, **HOME**. HOME makes the cursor go to the top left corner of the Debug Terminal. As you ran your program, it looked like the 0 changed to a 1 when you stalled the wheel. But really, each time through the DO...LOOP, HOME just makes the new value of stuck overwrite the last displayed value.

```
  DEBUG HOME, BIN stuck
```

This DEBUG command also uses a new formatter, **BIN**. BIN is short for binary, the number system that uses only 0s and 1s. Remember, stuck is a bit-sized variable that can only be equal to 0 or 1. If you forgot the BIN formatter, your Debug Terminal was probably blank when you ran the program.

This short PAUSE command gives your eyes time to read the Debug Terminal display, without constant flickering, before starting the loop over again.

```
  PAUSE 50
LOOP
```

Now that we know how to check the status of a sensor and store it in a variable, let's use that information to make a program decision.

Making Decisions with IF...THEN

The IF...THEN command allows a program to test a condition, and then use the answer to make a decision about what to do next. Testing a condition might mean to check whether a sensor is sending a high signal or a low signal. There are several syntax options for the IF...THEN command. For a simple instruction, you can put the whole thing on one line, like this:

IF *Condition* THEN *Statement*

Think of it like this:

IF *a wheel is stuck* THEN *play an alarm sound on the speaker*

The command would read like this:

```
IF stuck = 1 THEN FREQOUT Speaker, 100, 950
```

If you have a more complex decision to make, you can use this syntax which keeps it organized on several lines:

IF *Condition(s)* THEN
** *Statement(s)***
ELSE
** *Statement(s)***
ENDIF

Think of it like this:

IF *the wheels are free* THEN
  run the motors
ELSE
  stop the motors
  play an alarm on the speaker
ENDIF

In a program, the command would look like this:

```
IF stuck = 0 THEN
  PULSOUT MotorRight, 3000
  PULSOUT MotorLeft, 2500
ELSE
  PULSOUT MotorRight, 2000
  PULSOUT MotorLeft,  2000
  FREQOUT Speaker, 500, 440
  FREQOUT Speaker, 500, 880
  FREQOUT Speaker, 500, 440
  FREQOUT Speaker, 500, 880
ENDIF
```

It is very useful to use an IF ...THEN statement inside a DO...LOOP. This way, a program can test a condition over and over again, and decide which action to take after each test. Let's use this sample IF...THEN statement inside a DO....LOOP:

- √ Make sure your Scribbler is on its box with its wheels clear.
- √ Enter, save and run StallCircle.bs2, on the next page.
- √ Turn off the Scribbler and disconnect the programming cable.

StallCircle.bs2

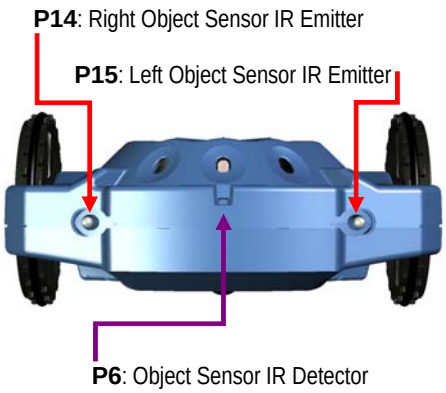
```
' Scribbler PBASIC Programming Guide - StallCircle.bs2
' Drive in circle, stop if stall sensor activates
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program running!"

' I/O PIN Definitions
Stall      PIN 7
Speaker    PIN 11
MotorRight PIN 12
MotorLeft  PIN 13

' Motor Initalization
LOW MotorRight
LOW MotorLeft
PAUSE 100

' Variable Declaration
stuck VAR Bit

DO
  stuck = Stall

  IF stuck = 0 THEN
    PULSOUT MotorRight, 3000
    PULSOUT MotorLeft, 2500
  ELSE
    PULSOUT MotorRight, 2000
    PULSOUT MotorLeft,  2000
    FREQOUT Speaker, 500, 440
    FREQOUT Speaker, 500, 880
    FREQOUT Speaker, 500, 440
    FREQOUT Speaker, 500, 880
  ENDIF
LOOP
```

√ Put the Scribbler on the floor in a clear area, and turn the power back on.
√ As the Scribbler drives in a circle, block it to trigger the stall sensor.
√ When the alarm sounds, move the obstacle so the Scribbler can drive again.

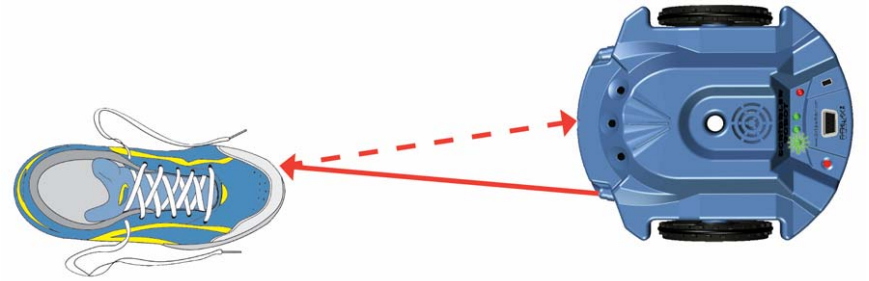
You have seen IF...THEN working in a DO...LOOP. IF the wheels are free (stuck = 0), THEN the motors get "go" commands. ELSE, when you block the Scribbler (stuck = 1) the motors get "stop" commands, and an alarm plays for two seconds. When the loop repeats, the Scribbler tries to drive again. If you have moved the obstacle, it will keep driving. If you have not moved the obstacle, the stall sensor will kick in and the motors will stop again.

Infrared Object Sensor System

Infrared is a color of light that the Scribbler can see, but your eyes cannot. The Scribbler Robot's object sensor system uses two infrared light emitter "headlights," and one infrared light detector between them.



The system works like this: Infrared light is emitted from the right IR emitter (an IR LED), flashing very rapidly in a special pattern. The IR detector in the middle looks for this light pattern bouncing off of objects. If it sees the infrared light signal reflected, it reports that there is an object on that side. Then, the right IR LED turns off, and the process is repeated with the left IR LED.



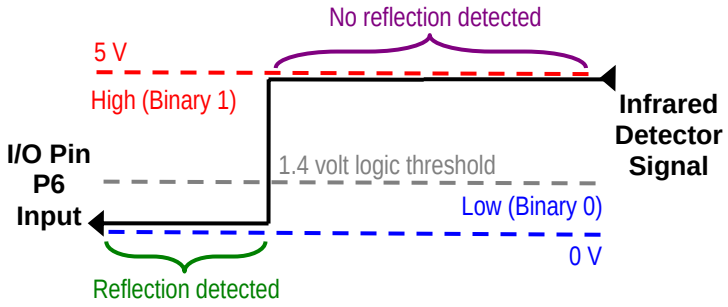
Infrared light reflects easily off of shiny, light-colored surfaces, but it is absorbed by dull, dark-colored surfaces. So, the infrared object detection system will be able to see a white rubber sneaker better than a black velvet slipper.

The Scribbler's infrared detector has a **filter** that allows it to see only infrared light flashing very rapidly, at 38,500 hertz (38.5 kilohertz). To make our infrared LEDs send out this **modulated** 38.5 kHz signal, we use the FREQOUT command.

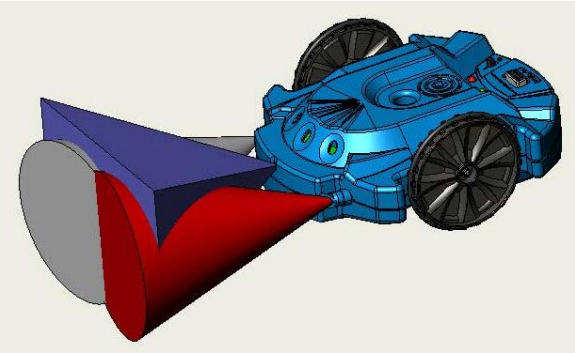
```
ObsTxRight PIN 14

FREQOUT ObsTxRight, 1, 38500
```

We use a FREQOUT *Duration* of 1 because we only want the IR LED to emit a short burst of light before the infrared sensor checks for a reflection.

The infrared detector is connected to I/O pin P6. If infrared light flashing at 38.5 kHz hits the detector, it sends a low signal to P6 (binary 0). If no such modulated infrared light hits the sensor, it sends a high signal to P6 (binary 1). As with the stall sensor, we can use a variable to remember the status of the infrared detector.



This illustration shows the beam of light from each IR LED as a cone. The infrared detector field of vision overlaps both cones. By sending a short burst of light from only one IR LED then checking the detector, you can tell if an object is on that side. This example code sequence will test to see if an object is detected in front of the Scribbler, to the right or to the left. It uses two variables, one for each IR emitter.



```
' I/O Pin Definitions
ObsRx          PIN 6           ' infrared detector
ObsTxRight     PIN 14          ' right IR emitter
ObsTxLeft      PIN 15          ' left IR emitter

' Variables
eyeRight VAR Bit               ' Variables to store sensor
eyeLeft  VAR Bit               ' status after using each IR LED

FREQOUT ObsTxRight, 1, 38500 ' Emit IR from right IR LED
eyeRight = ObsRx               ' put sensor status in eyeRight

FREQOUT ObsTxLeft, 1, 38500  ' Repeat test with left IR LED
eyeLeft = ObsRx                ' put sensor status in eyeLeft
```

Example Program: EyeTest.bs2

Let's use this example code in a complete program. This program behaves a lot like the factory program Demo Mode #2, Object Detection.

- √ Make sure your Scribbler is on its box with its wheels clear.
- √ Enter, save and run EyeTest.bs2, on the next page.
- √ Hold a piece of white paper about 6 inches in front of the Scribbler, and move it back and forth, while watching the green LEDs.

If the Scribbler sees the paper on the right, the right green LED lights up. If it sees the paper the left, the left green LED lights up. If the paper is directly in front, both LEDs will light up.

Infrared Interference from Light Fixtures

If the green LEDs light up when there is nothing in front of the Scribbler, you may have a light fixture that is emitting modulated infrared light at 38.5 kHz. This sometimes happens with overhead fluorescent lighting. Run EyeTest.bs2 and point your Scribbler at the light fixture to make sure. If the green LEDs light, up, you are detecting IR interference. Always turn off that light fixture whenever you are using the infrared object detection system, or you may get unexpected behavior from your Scribbler.

EyeTest.bs2

```
' Scribbler PBASIC Programming Guide - EyeTest.bs2
' Test infrared object detectors, report via green LEDs
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

' I/O Pin Definitions
ObsRx          PIN 6            ' infrared detector
LedRight       PIN 8            ' right green LED
LedLeft        PIN 10           ' left green LED
ObsTxRight     PIN 14           ' right IR emitter
ObsTxLeft      PIN 15           ' left IR emitter

' Variable Declarations
eyeRight VAR Bit                ' To remember sensor status
eyeLeft  VAR Bit                '  after each IR LED is used

' Main Program
DO
  FREQOUT ObsTxRight, 1, 38500 ' Emit IR from right IR LED
  eyeRight = ObsRx              ' put sensor status in eyeRight

  FREQOUT ObsTxLeft, 1, 38500  ' Repeat test with left IR LED
  eyeLeft = ObsRx               ' put sensor status in eyeLeft

  IF (eyeRight = 0) THEN        ' if object is detected at right
    HIGH LedRight               '  turn on right green LED
  ELSE                          ' else, no object is detected
    LOW LedRight                '  turn off right green LED
  ENDIF

  IF (eyeLeft = 0) THEN         ' if object is detected at left
    HIGH LedLeft                '  turn on left green LED
  ELSE                          ' else, no object is detected
    LOW LedLeft                 '  turn off left green led
  ENDIF
LOOP                            ' repeat test
```

How EyeTest.bs2 Works

This program uses five I/O pin definitions, three for object detection and two for the green LEDs.

```
ObsRx          PIN 6
LedRight       PIN 8
LedLeft        PIN 10
ObsTxRight     PIN 14
ObsTxLeft      PIN 15
```

Two variables are needed – one to store the IR detector status after using the right IR emitter, and one to store the IR detector status after using the left IR emitter. That way we can remember what the Scribbler sees (or doesn't see) on both sides at the same time.

```
eyeRight VAR Bit
eyeLeft  VAR Bit
```

After we have prepared the pin directives and initialization and declared the variables, we begin the main program loop. This first FREQOUT command causes the right IR LED to emit infrared light, modulated at 38.5 kHz, for 1 ms.

```
DO
  FREQOUT ObsTxRight, 1, 38500
```

If this modulated infrared light bounces off an object and hits the IR sensor, the sensor will send a low signal (binary 0) to the ObsRx pin. If not, the IR detector will send a high signal (binary 1). This next command will check to see if the ObsRx pin is reporting a 0 or a 1, and will store that value in the variable eyeRight.

```
  eyeRight = ObsRx
```

The next two commands repeat the same process using the left IR LED and the variable eyeLeft.

```
 FREQOUT ObsTxLeft, 1, 38500
 eyeLeft = ObsRx
```

Now the program uses the information stored in the variables to make a decision. If the IR detector saw a reflection after using the right IR LED (eyeRight = 0), then turn on the right green LED with the HIGH command. Otherwise, turn off the right LED with LOW.

```
  IF (eyeRight = 0) THEN
    HIGH LedRight
  ELSE
    LOW LedRight
  ENDIF
```

Next, the same decision-making happens using the value stored in eyeLeft, and the process repeats.

```
IF (eyeLeft = 0) THEN
    HIGH LedLeft
  ELSE
    LOW LedLeft
  ENDIF
LOOP
```

Your Turn

√ Modify EyeTest.bs2 so that different tones play when the Scribbler detects your hand on the right and on the left.

Putting it All Together

Let's try one final example program that runs all of the Scribbler Robots systems we have used so far. But first, here's one more programming trick that is useful when using more than one sensor in the same program.

IF...THEN with Logical Operators

In the last example program, we made the green LEDs work in tandem with the infrared object sensor. We used separate IF...THEN commands for each IR emitter and green LED pair. In each case, there was only one condition to test: If an object can be seen on this side, turn on the corresponding green LED. But sometimes, you may want something to happen only if several conditions are met at once.

Suppose you want the Scribbler to drive, but you want it to check for obstacles with both its infrared sensor and its stall sensor. We would want the Scribbler to stop driving if:

1. an object can be seen to the right

OR

2. an object can be seen to the left

OR

3. the motors are running but the wheels can't turn

OR is a **logical operator** that can be placed inside an IF...THEN command to make a complex condition. Using OR twice in a single IF...THEN command will cause the program to take the same action if any one of those three conditions is true:

```
IF (eyeRight = 0) OR (eyeLeft = 0) OR (stuck = 1) THEN
```

There are three other logical operators that can be placed inside IF...THEN commands: **AND**, **NOT**, and **XOR**. AND and OR work just the same way they do in ordinary speech. NOT can be thought of as "anything but that!" XOR is kind of confusing at first. It means "one thing or the other must be true, but not both, for the whole test to be true." Now, let's use OR in a program.

Example Program: SeeCircle.bs2

SeeCircle.bs2 is a bit like StallCircle.bs2, but it adds infrared obstacle detection. This program will make the Scribbler drive in a circle with its green LEDs on. If the Scribbler sees an obstacle, it stops driving, turns off the green LEDs and plays an alarm until you remove the obstacle. If the Scribbler hits an object that it cannot see and gets stuck, the stall sensor activates and stops the motors for one and a half seconds. Then the Scribbler tries to drive again.

- √ Make sure your Scribbler is on its box with its wheels clear.
- √ Enter, save and run SeeCircle.bs2, on the next page.
- √ Turn off the Scribbler and disconnect the programming cable.
- √ Put the Scribbler on the floor in a clear area, and turn the power back on.

SeeCircle.bs2

```
' Scribbler PBASIC Programming Guide - SeeCircle.bs2
' Drive Scribbler in circle with LEDs on.  If it sees an object: stop,
' turn off LEDs, and sound alarm until the obstacle is removed. If it
' stalls, stop and sound alarm for 1.5 sec. then try driving again.
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program running!"

' I/O Pin Definitions
ObsRx          PIN 6                ' infrared detector
Stall          PIN 7                ' stall sensor
LedRight       PIN 8                ' right green LED
LedCenter      PIN 9                ' center green LED
LedLeft        PIN 10               ' left green LED
Speaker        PIN 11               ' speaker
MotorRight     PIN 12               ' right wheel motor
MotorLeft      PIN 13               ' left wheel motor
ObsTxRight     PIN 14               ' right IR emitter
ObsTxLeft      PIN 15               ' left IR emitter

' Motor Initialization
LOW MotorRight
LOW MotorLeft
PAUSE 100

' Variable Declarations
eyeRight VAR Bit                    ' To remember IR sensor status
eyeLeft  VAR Bit                    '  after each IR LED is used
stuck    VAR Bit                    ' To remember stall sensor status

DO
  FREQOUT ObsTxRight, 1, 38500      ' Emit IR from right IR LED
  eyeRight = ObsRx                  ' put sensor status in eyeRight

  FREQOUT ObsTxLeft, 1, 38500       ' Repeat test with left IR LED
  eyeLeft = ObsRx                   ' put sensor status in eyeLeft

  stuck = Stall                     ' put stall sensor status in stuck

  IF (eyeRight = 0) OR (eyeLeft = 0) OR (stuck = 1) THEN  ' an object
    PULSOUT MotorRight, 2000         '  is in the way. Stop both of
    PULSOUT MotorLeft, 2000          '  the wheel motors...

    LOW LedRight                     ' ...turn off the LEDs...
    LOW LedCenter
    LOW LedLeft

    FREQOUT Speaker, 750, 1200, 1500  ' ...and sound an alarm.
    FREQOUT Speaker, 750, 800, 1200

  ELSE                              ' Else, no objects are in the way
    PULSOUT MotorRight, 2500        ' so right wheel forward, and left
    PULSOUT MotorLeft, 3000         ' wheel fwd faster, drive in circle

    HIGH LedRight                   ' and turn on the LEDs back on
    HIGH LedCenter
    HIGH LedLeft
  ENDIF
LOOP
```

- √ As the Scribbler drives in a circle, hold a piece of paper in front of it to trigger the obstacle detector system.
- √ Let it drive again, and then try to block it from the side to trigger the stall sensor.
- √ When the alarm sounds, move the obstacle so the Scribbler can drive again.

How SeeCircle.bs2 Works

The program uses I/O pin definitions for all of the Scribbler parts this program controls: the motors, the green LEDs, the IR LEDs, the speaker, the stall sensor, and the infrared detector.

```
ObsRx          PIN 6
Stall          PIN 7
LedRight       PIN 8
LedCenter      PIN 9
LedLeft        PIN 10
Speaker        PIN 11
MotorRight     PIN 12
MotorLeft      PIN 13
ObsTxRight     PIN 14
ObsTxLeft      PIN 15
```

Next comes motor initialization, setting their I/O pins to output low and allowing wake-up time.

```
LOW MotorRight
LOW MotorLeft
PAUSE 100
```

Three bit-sized variables are declared to hold sensor values: one for the IR sensor after using the right IR emitter, one for the IR sensor after using the left IR emitter, and one for the stall sensor.

```
eyeRight VAR Bit
eyeLeft  VAR Bit
stuck    VAR Bit
```

The loop begins with a short light burst from the right IR emitter. The I/O pin ObsRx is monitoring the IR detector signal (0 = reflection, 1 = no reflection) and the status of ObsRx gets stored in the variable eyeRight. The process repeats with the left IR emitter and eyeLeft.

```
DO
  FREQOUT ObsTxRight, 1, 38500
  eyeRight = ObsRx

  FREQOUT ObsTxLeft, 1, 38500
  eyeLeft = ObsRx
```

The Stall I/O pin is monitoring the stall sensor signal (0 = not stuck, 1= stuck). The next instruction stores the status of Stall in the stuck variable.

```
stuck = Stall
```

Now, it is time use the three variables to test if there is something in the Scribbler's way. IF the infrared sensor saw a reflection from the right (eyeRight = 0) OR a reflection from the left (eyeLeft = 0), OR if the stall sensor says the wheels are not turning (stuck = 1) THEN we want the motors to stop....

```
IF (eyeRight = 0) OR (eyeLeft = 0) OR (stuck = 1) THEN
    PULSOUT MotorRight, 2000
    PULSOUT MotorLeft, 2000
```

....and we want the green LEDs to turn off....

```
    LOW LedRight
    LOW LedCenter
    LOW LedLeft
```

...and we want an alarm to sound for one and a half seconds.

```
    FREQOUT Speaker, 750, 1200, 1500
    FREQOUT Speaker, 750, 800, 1200
```

But if the three variables all show that there is nothing in the Scribbler's way, it is safe to command the motors to drive forward....

```
  ELSE
    PULSOUT MotorRight, 2500
    PULSOUT MotorLeft, 3000
```

...and for the green LEDs to turn on. The decision-making ends, and the test to repeats:

```
    HIGH LedRight
    HIGH LedCenter
    HIGH LedLeft
  ENDIF
LOOP
```

You might have noticed something funny. If the Scribbler stops when it sees an object, it does not try to drive until the object is removed. But if it gets stuck, it tries to drive again after 1.5 seconds. This happens because the IR detector system keeps working when the motors stop, but the stall sensor only senses that the Scribbler is stuck when the motors are running but the wheels can't turn. So, after it senses a stall and the motors turn off, it cannot keep sensing that it is stuck until the wheels try to turn again.

Your Turn

√ Modify SeeCircle.bs2 so the IF...THEN command uses two AND operators instead of OR, but works the same way. HINT: take a close look at the variable conditions too.

Congratulations!

You have learned a lot! You have gained a lot of PBASIC programming skills. Now, you know how to:

- Write a PBASIC program and load it into the BASIC Stamp inside your Scribbler
- Create and format Debug Terminal messages
- Display your name using ASCII code
- Make actions repeat in an infinite loop
- Create and use I/O pin definitions
- Declare variables
- Use bit variables to store information from a sensor
- Test a variable condition to make a program decision
- Make a test that requires more than one condition to be met

NOW YOU KNOW

AND
BIN
Bit
CR
DEBUG
DEC
DO...LOOP
END
FREQOUT
HIGH
HOME
IF...THEN...ELSE
LOW
OR
PAUSE
PIN
PULSOUT
VAR

You have applied these skills in many ways to control your Scribbler robot.

- You know how the BASIC Stamp I/O pins are connected to your Scribbler.
- You know what three states these I/O pins can be in, and what they are for.
- You can switch the Scribbler's green LEDs on and off.
- You can make the speaker generate tones at different frequencies and durations to make alarm sounds or play a song.
- You can send control signals to the motors to make the Scribbler drive forward, backward, or turn.
- You can use a program test and calibrate the motors to make the Scribbler drive straight.
- You know that the BASIC Stamp I/O pins interpret high and low input signals by using the 1.4 volt TTL logic threshold.
- You can use an I/O pin as an input to monitor the status of a sensor, such as the stall sensor.
- You can coordinate the use of three I/O pins together with the infrared object detection system.
- You can use the Debug Terminal, the green LEDs and the speaker to let you know what a sensor is detecting.
- You can use the stall sensor and the infrared obstacle sensor system with the motors to make the Scribbler respond to its environment as it drives.

Did you have any idea that you had learned so much? Great Job! Now, you are ready to experiment with writing your own custom PBASIC programs for your Scribbler Robot!

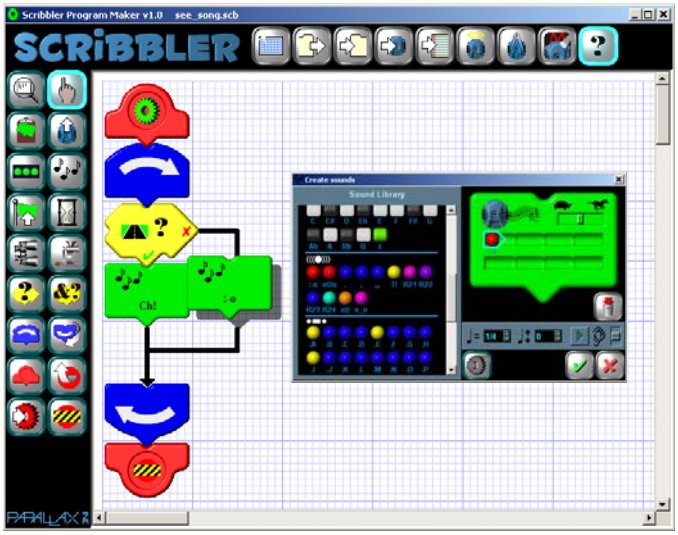
More about the Scribbler PBASIC Programming Guide

The *Scribbler PBASIC Programming Guide* is a series of educational modules that introduce robotics and programming concepts in an illustrated, step-by-step format.

This module, Writing Programs, was designed to help the beginner learn to use BASIC Stamp Editor through building simple programs. It did not introduce all of the capabilities of the BASIC Stamp microcontroller or the Editor software, nor did it use all of the features of the Scribbler Robot. Watch our website, www.ScribblerRobot.com, for more modules in this series.

Scribbler Forums

Would you like to share your experiences programming your Scribbler in PBASIC with others? Visit our moderated forums at http://forums.ScribblerRobot.com where you can post questions, help others, and share ideas with other Scribbler owners.



Graphical Programming

Would you like to learn to program your Scribbler Robot using graphics instead of text? The Scribbler Program Maker GUI (Graphical User Interface) software lets you build programs graphically with action tiles and mouse clicks. This is the perfect starting place for beginning programmers age 8 and up.

You can download the latest versions of the Scribbler Program Maker software and the *Scribbler GUI Programming Guide* free from the Downloads page of our website www.ScribblerRobot.com.

Have a Great Idea?

Did you design a really great Scribbler PBASIC project that you would like to share? We are inviting educators and robotics enthusiasts to write modules for the *Scribbler PBASIC Programming Guide*. If you are interested, email aalvarez@parallax.com.

ASCII Chart (Characters 32 through 127)

ASCII Codes 32 through 126 correspond to the characters that can be displayed with the BASIC Stamp Editor's Debug Terminal (127 is "delete").

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	20	space	56	38	8	80	50	P	104	68	h
33	21	!	57	39	9	81	51	Q	105	69	i
34	22	"	58	3A	:	82	52	R	106	6A	j
35	23	#	59	3B	;	83	53	S	107	6B	k
36	24	$	60	3C	<	84	54	T	108	6C	l
37	25	%	61	3D	=	85	55	U	109	6D	m
38	26	&	62	3E	>	86	56	V	110	6E	n
39	27	'	63	3F	?	87	57	W	111	6F	o
40	28	(	64	40	@	88	58	X	112	70	p
41	29	)	65	41	A	89	59	Y	113	71	q
42	2A	*	66	42	B	90	5A	Z	114	72	r
43	2B	+	67	43	C	91	5B	[	115	73	s
44	2C	,	68	44	D	92	5C	\	116	74	t
45	2D	-	69	45	E	93	5D	]	117	75	u
46	2E	.	70	46	F	94	5E	^	118	76	v
47	2F	/	71	47	G	95	5F	_	119	77	w
48	30	0	72	48	H	96	60	`	120	78	x
49	31	1	73	49	I	97	61	a	121	79	y
50	32	2	74	4A	J	98	62	b	122	7A	z
51	33	3	75	4B	K	99	63	c	123	7B	{
52	34	4	76	4C	L	100	64	d	124	7C	|
53	35	5	77	4D	M	101	65	e	125	7D	}
54	36	6	78	4E	N	102	66	f	126	7E	~
55	37	7	79	4F	O	103	67	g	127	7F	delete

Dec = Decimal (base 10) Hex = Hexadecimal (base 16) Char = Character