1. Introduction

Overview
The BASIC Stamp is the simplest microcontroller to learn about and to use, even for the novice who has never programmed before. This guide is for students in ME 2011, or students anywhere who are confronting the Stamp for the first time. For advanced Stamp users, prowl the web; there are lots of resources.

The BASIC Stamp is a microcontroller made by Parallax Inc (www.parallaxinc.com). The Stamp microcontroller chip comes in several flavors and Parallax also produces several packaged Stamp boards that incorporate the Stamp chip plus some peripherals. The Parallax web site has details on the complete Stamp product line.

This guide covers the BASIC Stamp HomeWork Board (#28158), a product introduced by Parallax in 2002 exclusively for students and educators. With the HomeWork Board, you can write programs and create interface circuits to read switches and other sensors, and to control motors and lights with very little effort. Many of the pictures and drawings in this guide were taken directly from the documentation on the Parallax Web site, the place to turn if you need more information. Although we cover the HomeWork board, much of the material in this guide is appropriate for other Stamp models. The Stamp Interfacing section on the ME 2011 web site (www.me.umn.edu/courses/me2011) covers more on interfacing the Stamp to the real world.

The HomeWork Board features a BASIC Stamp 2 controller module with a PIC16C57C-20/SS microprocessor for running programs and a 24LC16B 2K EEPROM for storing programs. It is mounted on a board with a LM2936 5V, 50 mA voltage regulator that provides a voltage source to the chips, a female DB-9 serial port connector for talking to the host PC, power and ground connections, a breadboard area for adding your own circuits, and clips for attaching a 9V battery to power the board. The I/O pins on the Stamp have 220 ohm resistors attached which means it is practically impossible to damage the board through wiring mistakes. An important feature of the Stamp is that you can create a control program on the host PC, download it to the Stamp and it will run automatically. Remove the serial cable connection to the PC, and the program will still run from the top each time you push the reset button. Remove the battery and put the HomeWork board in a closet for six months. When you reconnect the battery, the last program you stored will run. This means that you connect the board to the host PC to develop and debug your program, but once that is done, you no longer need the PC to run the program. The BASIC Stamp code speed is 4000 instructions/sec which means 500-600 PBASIC code lines every second.
Here is a picture of the HomeWork Board

And here is a schematic of the HomeWork with a description of its parts taken from the Parallax HomeWork Board documentation.

A. BASIC Stamp 2 Module. The BASIC Stamp is comprised of several components: PIC16C57-20/SS- a Microchip 2K 8-bit microcontroller programmed with the BASIC Stamp “Interpreter” that runs the Parallax BASIC Stamp code like a mini operating system; the 24LC16B EEPROM - a Microchip 2KEEPROM (electrically erasable programmable read-only memory) with a user-modifiable read-only memory (ROM) that can be reprogrammed with your BASIC Stamp code; a 20 mHz resonator to provide an accurate clock source for the BASIC Stamp; 220 Ohm resistors across the I/O pins to provide current protection for mistake wiring; an LM2936 voltage regulator which provides 50 mA for the BASIC Stamp and your circuits powered from the breadboard’s Vdd.
B. DB-9 Female. The DB-9 port connects via serial cable to your PC’s serial port. This port is used to download programs to your BASIC Stamp and for the BASIC Stamp to send data back to the PC.

C. Power and Ground Connections. Vdd is regulated 5V, Vin is 9V from the transistor battery, Vss is ground.

D. Breadboard. Two areas of 5 column x 17 row breadboard project area. Connections are horizontal separated by the trough. Sets of five holes in each row are connected electrically.

E. Power Button. Illuminated when the BASIC Stamp is running a program only.

F. Reset Circuit. Reset the BASIC Stamp by pressing this button.

G. Battery Tie-Downs. If using the HomeWork Board Through-Hole Version for projects involving high-vibration (on R/C airplanes, robots, etc.) a “zip-tie” can hold the battery firmly to the board if looped through these holes.


What you need for a working system

1. Stamp HomeWork Board
2. 9V battery
3. Serial port cable (standard, straight through; not a null modem cable)
4. Host PC running the Stamp application (stampw.exe) under some flavor of Windows. Stampw.exe is available from the Parallax web site.

There are several flavors of the Stamp editor on the Parallax web site. For this guide, we'll assume you have the Basic Stamp 2 editor for Windows ver. 1.1, stampw.exe. Stampw.exe is not the newest version, but it works fine, is small in size and doesn’t take long to download.

Have a Mac? Go to the parallax.com web site and look for the “Software for Macintosh” section and download the Mac version of the Stamp editor.

What to do if your computer has no serial port

The Homework board connects to your PC through a serial port. Many modern PCs, particularly laptops, may not have a serial port but will have plenty of USB ports. To solve this problem, buy yourself a USB to Serial converter that creates a virtual serial port on the PC. There are many converter models and most (but not all) will run with the Stamp. Price does not matter and converters can be found for as low as $10. A $15 model from Parallax (PN 28030) is guaranteed to work with the Stamp but requires a separate USB A to mini B cable (Parallax PN 805-00006, $6.95.).

For students in ME 2011, a usb serial adapter is included in the Robot Kit. Instructions for installing the adapter are in the Computer Connections section of the robot project web site.
**Getting started**

Plug the 9V battery into the battery terminals. **WARNING:** Check polarity *before* making the connection; otherwise you might blow out your board. The onboard LED (just above the reset button) may or may not light. This LED goes on when the Stamp is executing a program. If there was already a program burned into the Stamp from before the LED will be on, if there is no program it will be off.

**Warning:** Do not put your board down on a conductive surface; you will short out the pins on the back!

Connect the serial cable to the Stamp board, and the other end to the serial port connector on your PC. This connector is generally on the back, and will be the only connector that the female end of the cable will fit into.

Start the stampw.exe program.

In the editing window that comes up, enter the following one line program:

```
  debug "I'm alive"
```

Your editor window will look something like this

![Editor Window with Program](image)

By selecting Run from the menu bar, then Run (or click on the right-facing triangle on the tool bar, or simply hit Ctrl+R) the program will be downloaded to the Stamp and run.
If all goes well, a Debug window will pop up with your message looking something like this:

![Debug Terminal](image)

Congratulations; you have created and run your first BASIC Stamp program!

Push the Stamp reset button a few times and see what happens.

**Troubleshooting**

If you get a "BASIC Stamp II detected but not responding on COMn! Check power supply." or similar error message, or if the program runs but no message pops up, try the following:

- Run the Stamp program again
- Check that the battery is all the way in.
- Check that the serial cable is secure at both ends.
- Measure your battery voltage. If it's less than 7 volts while connected to the Stamp, get yourself a new battery.
- Reboot your PC because sometimes the serial port can lock up
- Try killing all programs on your PC except Stamp.exe (Ctrl+Alt+Delete, End all running tasks except Explorer, Systray and stampw)
- Ask a friend for help

If you get an error that's something like "Error: Expected xxxxxx", you have a typo in your code. The code will be highlighted somewhere on or after the bug; it's up to you to search it out and correct.
**Flashing an LED**

Light emitting diodes (LED's) are handy for checking out what the Stamp can do. For this task, you need an LED, a 330 ohm resistor, and one short piece of 22 or 24 g wire, stripped about 0.25" at each end. The figure to the right is a sketch of an LED and its symbol used in electronic schematics.

Connect the notched or flat side of the LED (the notch or flat is on the rim that surrounds the LED base; look carefully because it can be hard to find) to Vss (ground) and the other side to a hole in the white breadboard. Connect one end of the resistor to the LED and the other end to a wire. Touch the free end of the wire to Vdd (5V) and the LED should light. If not, try changing the orientation of the LED. Now connect the free end of the wire to P0. Your setup should look something like the figure on the left. On the white breadboard, any group of 5 holes in a horizontal line is connected electrically on the breadboard. To "wire" two components together, place anywhere in the same horizontal row.

On the LED, current runs from the anode (+) to the cathode (-) which is marked by the notch. The circuit you just wired up is represented in schematic form in the figure to the right.

Create and run the following program

```plaintext
high 0
pause 1000
low 0
```

What happened? Push the reset button to run the program again.

Now try this program which will flash the LED at 1.0 Hz. The lines beginning with an apostrophe are comments, always good to add to a program.

```plaintext
'---------------------------------------------------
' Stamp LED flasher. 1.0 Hz on Pin 0. (wkd 9/20/07)
'---------------------------------------------------
loop:
    high 0             ' pin 0 high (LED on)
    pause 1000
    low 0
```

If you want to check code syntax without a Stamp connected, use Ctrl+T (or Run > Check Syntax, or use the toolbar button).

If you want to see how much memory your program takes up, use Ctrl+M (or Run > Memory Map, or use the toolbar button).
pause 500  ' wait 500 msec
low 0     ' pin 0 low (LED off)
pause 500  ' wait 500 msec
goto loop ' loop forever

The "high 0" command says to set Pin 0 (P0) of the Stamp to high, or +5 volts. This sends current from the pin, through the resistor, through the LED (which lights it) and to ground. The "pause 500" command waits for 500 msec. The "low 0" command sets Pin 0 to 0V stopping the current thereby turning the LED off. The "goto loop" command tells the program to jump to the label "loop:" which in this case makes the cycle start all over again.

Here's a shorter way to do the same thing.

'---------------------------------------------------
' Stamp LED flasher, version 2 (wkd 9/20/07)
'---------------------------------------------------

loop:
toggle 0     ' change state of pin
pause 500    ' wait 500 msec
goto loop    ' loop forever

The "toggle" command changes the state of an output pin. If the pin was high, toggle turns it off, and if off, toggle turns it back on.

**Reading a switch**

The LED exercise shows how to have the Stamp control the outside world. Many applications require reading the state of sensors, including switches. For this exercise, you will have the Stamp read the state of a normally-open push button switch and display the results on the PC using the debug command.

You will need a switch, a 10 kohm resistor and some pieces of 22 g hookup wire. If you don't have a switch, substitute two wires and manually connect their free ends to simulate a switch closure. The figure to the right shows a picture of a pushbutton switch and its schematic symbol. Note that the symbol represents a switch whose contacts are normally open, but then are shorted when the button is pushed.

Wire up the circuit shown in schematic form to the left. Then create and run this program

```
check: debug dec in1
pause 250
goto check
```
When the switch is open, you should see a train of 1's on the debug screen. When closed, the 1's change to 0's. On the hardware side, when the switch is open, no current flows through the resistor. When no current flows through a resistor, there is no voltage drop across the resistor which means the voltage on each side is the same. Thus, in our circuit, when the switch is open, pin 1 is at 5 volts which the computer reads as a 1 state. When the switch is closed, pin 1 is directly connected to ground which is at 0 volts. The computer reads this as a 0 state.

Now try this program which is an example of how you can have the computer sit and wait for a sensor to change state.

```
'---------------------------------------------------
'  Stamp switch checker (wkd 09/20/07)
'---------------------------------------------------

check1:
  if in1 = 1 then check1
  debug cls, "somebody closed the switch!"
check2:
  if in1 = 0 then check2
  debug cls, "the switch is now open!"
goto check1
```

2. Stamp hardware

The power of the Stamp is not its ability to crunch code, but rather its ability to interact with the outside world through its input-output (I/O) pins. The HomeWork Board Stamp has 16 I/O pins (labeled P0 through P15) that can be used to turn motors and lights on and off and read the state of switches. Connections are made to the I/O pins by 22 gage wire jumpers running between the pins and the breadboard area.

Each I/O pin can sink or source about 20 mA of current, but the whole Stamp can source or sink no more than 50 mA. This is more than adequate for interfacing to most devices, but does mean that interface circuits are needed to control devices other than simple LED's. In other words, you cannot run a motor directly using the current available from a Stamp pin, but rather must have the Stamp pin drive an interface circuit that in turn drives the motor. A later section of this document shows how to interface to a small motor.

To interact with the outside world, the program sets ports (I/O pins) to high and low values via the PBASIC high and low instructions. The Stamp board sets the I/O pins to +5 V or 0 V accordingly. The switched 5 volt signal is used by the interface circuit. This sequence of events is shown below
To determine the state of switches and other sensors, the Stamp is able to read the voltage value applied to its pins as a binary number. The interface circuitry translates the sensor signal into a 0 or +5 V signal applied to the I/O pin. Through a program command, the Stamp interrogates the state of the pin. If the pin is at 0 V, the program will read it as a 0. If it is at +5 V, the program will read it as a 1. If more than +5 V is applied, you will blow out your Stamp board, so be careful.

The sequence of events to read a pin is shown below
Interacting with the world has two sides. First, the designer must create electronic interface circuits that allow motors and other devices to be controlled by a low (1-10 mA) current signal that switches between 0 and 5 V, and other circuits that convert sensor readings into a switched 0 or 5 V signal. Second, the designer must write a program using the set of Stamp commands that set and read the I/O pins. Examples of both can be found in the "Stamp Interfacing" document as well as in the sources given in the Resources chapter of this guide.

When reading inputs, pins must have either 0 or 5V applied. If a pin is left open or "floating", it will read random voltages and cause erratic results. This is why switches always have a 10K pull up resistor connected when interfacing to a Stamp pin.

3. Programming concepts

This chapter covers some basic concepts of computer programming, going under the assumption that the reader is a complete novice.

A computer program is a sequence of step-by-step instructions for the computer to follow. The computer will do exactly what you tell it to do, no more no less. The computer only knows what's in the program, not what you intended. Thus the origin of the phrase, "Garbage in, garbage out".

The set of valid instructions comes from the particular programming language used. There are many languages, including C, C++, Java, Ada, Lisp, Fortran, Basic, Pascal, Perl, and a thousand others. The BASIC Stamp uses PBASIC, a simplified variation of the Basic programming language.

For any programming language, the instructions must be entered in a specific syntax in order for the computer to interpret them properly. Typically, the interpretation is a two step process. A compiler takes the language specific text you enter for the program and converts it into a machine readable form that is downloaded into the processor. When the program executes, the processor executes the machine code line by line.

Basics of programming languages

All sequential programming languages have four categories of instructions. First are operation commands that evaluate an expression, perform arithmetic, toggle states of i/o lines, and many other operations. Second are jump commands that cause the program to jump immediately to another part of the program that is tagged with a label. Jumps are one way to break out of the normal line-by-line processing mode. For example, if you want a program to repeat over and over without stopping, have the last line of the program be a jump command that takes the program back to its first line. Third are branch commands that evaluate a condition and jump if the condition is true. For example, you might want to jump only if a number is greater than zero. Or, you might want to jump only if the state of an i/o line is low. Fourth are loop commands that repeat a section of code a specified number of times. For example, with a loop you can have a light flash on and off exactly six times.
Most programming languages contain a relatively small number of commands. The complexity of computers comes from combining and repeating the instructions several million times a second.

Here's a generic program.

1. Do this
2. Do that
3. Jump to instruction 6
4. Do the other thing
5. All done, sleep
6. If switch closed, do that thing you do
7. Jump to instruction 4

The computer will execute this line by line. The art of programming is simply a matter of translating your intent into a sequence of instructions that match.

Here is an example of a loop command followed by a branch command that uses an IF statement

```
for b0 = 1 to 6
    instructions
next b0
if b2 > 4 then label
    instructions
```

The commands inside the loop will be repeated six times. Following this, if the value of b2 is greater than 4, the program will skip to the instruction tagged with the specified label, and if not the line following the if statement will be executed.

Subroutines are a way of calling a section of code from a number of different places in the program and then returning from that section to the line that follows the calling line. Here's an example. The lines ending in ':' are labels used for jump, branch and subroutine addresses.

```
gosub apples
    instructions
end
apples:
    instructions
return
```

The subroutine is everything between the "apples" label and the RETURN instruction. When the program gets to the return, it jumps back to the line following the GOSUB.
Data sizes
When working with a microcontroller that interacts with the real world, you have to dig a little below the surface to understand numbering systems and data sizes.

A binary (base 2) variable has two states, off and on, or 0 and 1, or low and high. At their core, all computers work in binary since their internal transistors can only be off or on and nothing between. Numbers are built up from many digits of binary numbers, in much the same way that in the base 10 system we create numbers greater than 9 by using multiple digits.

A bit is one binary digit that can take on values of either 0 or 1. A byte is a number comprised of 8 bits, or 8 binary digits. By convention, the bits that make up a byte are labeled right to left with bit 0 being the rightmost, or least significant bit as shown below:

```
  b7  b6  b5  b4  b3  b2  b1  b0
```

Thus, in the binary number 011, bits 0 and 1 are 1 while bit 2 is 0. In the binary number 1000001, bits 0 and 7 are 1 and the rest are zero.

Here are a few binary to decimal conversions for byte size numbers.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000011</td>
<td>3</td>
</tr>
<tr>
<td>00000111</td>
<td>7</td>
</tr>
<tr>
<td>11111111</td>
<td>255</td>
</tr>
</tbody>
</table>

In a computer, variables are used to store numbers. A bit variable can take on two values, 0 and 1, and is typically used as a true/false flag in a program. A byte variable can take on integer values 0-255 decimal while a word variable (16 bits) can take on integer values 0-65,535. In simple microcontrollers like the Stamp, numbers can be positive valued integers only, no floating point and no negative numbers.

4. BASIC Stamp architecture

The Stamp has two kinds of memory, EEPROM for storing programs and RAM for storing data in variables used by your program. EEPROM has the nice property that it retains information when the battery is removed. After your program is loaded, you can disconnect the Stamp from the computer, remove the battery, and the program will run from the first instruction line each time you connect the battery. EEPROM is not used for all memory because it is relatively slow to read and write to and because it has a limited number of read/write cycles.

The Stamp has 32 bytes of RAM for storing variables. The first six bytes are reserved for the i/o pins and other system variables leaving space for 26 byte size variable that you can use in your program. These variables are prenamed b0 through b25, but you can change the names as shown later. You can also have word size (16 bit) variables that are...
prenamed w0 through w12. These overlap the byte size variable space, but if you use the variable declaration methods described later, you won't run into problems.

The Stamp has 16 input/output (I/O) pins for interfacing to the real world. On powerup, these pins are set to an input state, but are easily changed to outputs as described below.

5. PBASIC programming language
PBASIC is a simplified version of BASIC, one of the most popular programming languages in the world (thanks to Microsoft Visual BASIC that runs on PC's). PBASIC has just a few instructions and is limited to positive integer arithmetic, but adds special instructions to manipulate and read the I/O pins. In this guide, we will only cover the subset of PBASIC that is most useful to the novice Stamp designer. For more information on PBASIC, see the full Stamp manual available as a free download from the Parallax web site.

All PBASIC instructions are one line. The HomeWork Board can hold a program up to about 500 lines, has space for about 26 byte size variables and executes programs at about 4,000 lines per sec.

Creating a program
Programs are created in the Stamp editor and then downloaded to the Stamp board. Code must be entered in the proper syntax which means using valid command names and a valid grammar for each code line.

The editor will catch and flag syntax errors before download. Sometimes the error message can be cryptic and you have to do a bit of hunting because the actual error occurred before what was flagged.

Although your program may pass cleanly through the syntax checker, it still might not do what you wanted it to. Here is where you have to hone your skills at code debugging. The Stamp did what you told it to do rather than what you wanted it to do. The best way to catch these errors is to read the code line by line and be the computer. Having another person go through your code also helps. Skilled debugging takes practice.

Program formatting and syntax
Programs are entered line by line. Code is case insensitive which means "mylabel" is the same is "MyLabel". You can use case to make the code more readable, but the simplest is just to make everything lowercase.

Comments begin with an apostrophe ('). Everything after the ' is ignored.

Multiple instructions can be entered on the same line by separating them with colons. For example
Be cautious about jamming too much on one line. You don't save any program space this way, just paper on program printouts.

**Constants** are fixed numbers and can be entered as ordinary decimal numbers (integer only) or in hexadecimal (base 16) or in binary (base 2) as shown in the table below.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>$64$</td>
<td>$%01100100$</td>
</tr>
</tbody>
</table>

**Address labels** are used to reference locations in your program. They can be any combination of letters, numbers and underscore (_), but the first character must be a letter. When used to mark a location, follow the label with a colon. When referring to an address label in an instruction line, don't use the colon. Here's an example:

```plaintext
start:  toggle 0
        pause 1000
        goto start
```

**Variables** are allocated with the "var" command and can be either byte (8-bit) or word (16-bit). Byte size variables can hold positive integers in the range 0-255 while word size variables can be 0-65535. Use 8-bit variables unless you need to hold or calculate numbers larger than 255. You must explicitly allocate all of your variables before first use, so the variable allocation lines generally should go at the beginning of your program.

```plaintext
i      var byte
length var byte
width  var byte
depth  var byte
volume var word
```

Variable names can be any combination of letters and numbers but must start with a letter. Names reserved for PBASIC programming instructions cannot be used for variable names and will give you a "Error: Symbol is already defined" message.

If the concept of naming and allocating variables seems like too much, you are always free to use the predefined variables. These are b0 through b25 for byte variables and w0 through w12 for word variables. There is overlap, however, between the two. For example, w0 takes up the same memory locations on the computer as b0 and b1 and you will get unexpected results if you use both b0 and w0 in one program. Using named variables avoids this conflict.

**Symbols** are used to redefine how something is named and can be handy for making the code more readable. Symbols are defined with the "con" command and lines defining symbols should go at the beginning of your program. Here's an example without symbols for the case where an LED is connected to pin 0.
repeat: high 0  'LED on
pause 100
low 0  'LED off
pause 800
goto repeat

and here's the same using a symbol to define "LED"

LED con 0
repeat: high LED
pause 100
low LED
pause 800
goto repeat

Note how the proper use of symbols reduces the need for comments. Symbols are extremely useful to define for devices connected to pins. If you ever have to change the pin that the device connects to, you only have to change the single symbol definition rather than going through your whole program looking for references to that pin.

Math
The Stamp has positive integer arithmetic only. You can't enter 23.2 (that's a decimal) and you can't calculate 10 - 12 (that would give you a negative number). Also, if you have byte variables, no number, nor the result of any math operation can fall outside the range of 0 to 255. You can divide numbers, but the result will be truncated (not rounded) to the nearest integer. Thus in Stamp arithmetic, 17/3 = 5, and not 5.666 and not 6. Math operations are performed strictly in a left-to-right order. You can add parenthesis to group operations.

For advanced Stampers: Strictly speaking, the Stamp will keep proper track of negative numbers during add, subtract and multiply using the rules of two's complement, and will display negative numbers using the Debug command. The Stamp also has a few math functions you can use. Consult the Stamp manual for details.

The table below shows some of the valid math operators. Full details of their use can be found in the BASIC Stamp Manual.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>//</td>
<td>modulus (division remainder)</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left bit shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right bit shift</td>
</tr>
<tr>
<td>&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**The simple commands**

This section covers those very few commands you need to make the Stamp do something useful. These commands appear in order of priority. In fact, you can make a great machine using only input, output and pause instructions. Learning all the commands here will take you to the next level. And, if you need more, consult the BASIC Stamp Manual for full descriptions of the complete command set.

**debug**

The debug command lets you see what's going on inside the Stamp from your computer. For example, you can see the result of a math operation to determine if you are getting the right number. Or, you can see the state of an input pin to see if the Stamp is reading the state of a sensor or switch properly. When your interface circuits or program does not seem to be working, use the debug command to shed a little light on the situation. For debug to show anything, you need to have the host PC connected with the serial cable.

Here is a one line program to check if your Stamp board is alive and properly connected to the PC

```
debug   "I'm alive"
```

Here is a program that loops in place, displaying the value of an I/O pin. Very useful for checking the state of sensors or switches. Hook up your sensor and make it go on and off (or open and close your switch). This program will tell you if the Stamp is able to read the sensor properly. Try it out on your Stamp. After running, use a jumper wire to alternately connect pin 1 to Vdd (+5V) and Vss (0V).

```
repeat: debug cls, dec in1
        pause 100
        goto repeat
```

Some notes about the above example. The "cls" clears the screen so that you get a new value each time. The "dec" says to print the value in decimal. "in1" is the notion used when reading a pin.

If you wanted to see the states of pins 1 and 2 at the same time, you can chain them all in one debug command, adding some text notes to make it easier to read.

```
repeat: debug cls, "pin1 = ", dec in1, " pin2 = ", dec in2
        pause 100
        goto repeat
```

You may have noticed when trying this out that if you leave one of the pins disconnected, its state follows the other. This is because a pin left floating has an undefined value and will wander from high to low. So, use two jumper wires when trying out this example.

Here's one that checks the value of a variable after an addition
x  var  byte
y  var  byte
z  var  byte

x = 21
y = 20
z = x + y
dec z

debug "z = ",
The debug command has many more options that you can read about in the BASIC Stamp manual.

**high, low, toggle**
These commands turn I/O pins high (+5V) and low (0V) and are the workhorse commands for commanding the outside world of lights, motors, and anything else interfaced to your Stamp. They automatically configure the pin as an output. The command is followed by the number of the pin to turn high or low

```plaintext
high 0  'changes pin #0 to +5 volts
low 12  'changes pin #12 to 0 volts
```

The toggle command changes the state of a pin. If it was high, toggle turns it low. If low, toggle turns it high

```plaintext
toggle 5  'change state of pin 5
```

**pause**
Pauses the program for specified number of milliseconds. Since most interactions with the world involve timing, this is an essential instruction. Can pause for 0 to 65,535 millisec.

```plaintext
high 1    'turn motor attached to pin 1 on
pause 1000  'wait for 1 second
low 1     'turn motor off
```

**input, output**
These are commands used at the top of your program to define which I/O pins are being used as inputs to read the state of switches and sensors, and which are outputs to drive motors and lights. For example, if pin 0 is connected to an LED and pin 1 is reading the state of a pushbutton switch, you would have these lines at the start of your program

```plaintext
' Define pin i/o configuration
output 0  'led
input 1    'switch
```

On powerup, all pins default to inputs. The high and low commands automatically change pins to outputs, so if your program only uses high, low and pause, you don't need the input and output commands.
### if...then

This is the basic conditional branch instruction that allows your program to do two different things depending on whether a specified condition is true or false.

Here is one way to have your program wait in place until a switch is closed. Connect one end of a 10K resistor to Vdd (+5V) and the other to pin 1. Also connect that second end to one side of a pushbutton switch. Connect the other side of the pushbutton switch to Vss (0V). If you don't have a switch, just make one up by touching two wires whose ends are bare. Run this program then try closing the switch

```
    here: if in1 = 1 then here
          debug "The switch closed"
```

The if..then line reads the state of pin 1. If it is 1, which it will be for this circuit when the switch is open, the code jumps to the line marked by the label "here", which means the code will try the condition again and again at a rate of about 4000 times each second. When you close the switch, 0V is applied to pin 1 and its state is no longer a logical one. The next time the if…then statement is run, the condition is no longer true and the line following the if…then is executed.

The syntax for the if..then statement is always

```
    if condition then address
```

If the condition is true, the program will jump to the line marked by the label "address". If false, the line following the if…then is executed.

The condition compares one thing to another. In the example above, the state of pin 1 was compared to 1 with an equality condition. Other conditional operators are <> (not equal to), > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to).

You can have the program branch depending on the value of a variable. For example, this program will run the loop 30 times then branch

```
x    var byte

    x = 0
    repeat: x = x + 1
            debug dec x, cr
            if x <= 30 then repeat
            debug "all done"
```

Note the use of "cr" in the first debug statement to have each number appear on a new line in the debug window.

### for...next

18
The for...next pair of instructions is used to create program loops. Loops are useful when you want a chunk of code to be repeated a specified number of times. A variable is used to count the number of times the code is repeated. Here is an example that flashes an LED attached to pin 0 five times

```plaintext
n var byte
for n = 1 to 5
    high 0
    pause 500
    low 0
    pause 500
next
```

Note how the repeated code is indented to make it clear that this is a loop. Here's another way of achieving the same flash function but with fewer code lines

```plaintext
n var byte
for n = 1 to 10
    toggle 0
    pause 500
next
```

You can have the loop counter increment by two or by three or by any increment you like using the 'step' option. Here's a code sample you can run

```plaintext
j var byte
for j = 1 to 15 step 3
    debug "loop number ", dec j, cr
next
```

Loops can be nested within loops. This example will flash that LED 100 times (note that you have to go through the inner loop twice for one on/off flash cycle)

```plaintext
j var byte : k var byte
for j = 1 to 10
    for k = 1 to 20
        toggle 0
        pause 100
    next
next
```

**goto**

The goto statement commands the computer to jump immediately to another part of the program marked by an address label. One common use when the computer finishes the program and you want it to start all over again at the top as shown here

```plaintext
start:  'start of program
...program lines go here...
goto start  'jump back and do it all over again
```
Programs structured in this way will keep running over and over again until the battery is disconnected. When the battery is reconnected, the program will start once more at the top. Use as few goto's as you can because they make the program hard to read.

end

The end statement is used to mark the last line of your main program and causes the computer to cease executing program lines. Without an end statement, the computer keeps going and will execute whatever random code happened to be in memory during the last program stored. End statements aren't needed if you have a goto line at the bottom that causes the program to start all over again.

gosub, return

Subroutines are a powerful programming feature that are used when you want to set up a utility function that can be called from several places in the program. For example, let's say you wanted that LED connected to pin 0 to flash 3 times as an alert, but that you need to enable that alert at three different places in the program. One solution would be to type in the flashing code at the three separate program locations. This uses up precious code space and also means that if you change the flash function (for example, changing from 3 flashes to 4), you have to change the code in three places. A better solution is to write the flash function as a subroutine and to call it from the main body of the code using the gosub statement. Here's an example

```
j var byte
start: gosub alert           'run the alert
      ...program lines go here...
      gosub alert           'run the alert again
      ...more program lines go here...
      gosub alert           'and again
      end                   'main program is all done

alert:  for j = 1 to 6
       toggle 0
       pause 200
next
return
```

Several things should be noted here. When the computer encounters a gosub command, it immediately jumps to the address specified in the gosub statement and starts executing the commands in the subroutine. The subroutine ends with a return statement which tells the computer to go back to the program line that immediately follows the most recent gosub. It is this feature that allows you to call the subroutine from several different places in the code. The main program concludes with an end statement. Without this, the code would keep on going right through to the subroutine which you don't want. It is common practice to place all of your subroutines after the main program, but this means the last line of the main program has to be either an end or a goto statement to prevent the code from accidentally falling into the subroutines. Finally, it is fine to call one subroutine while in another. You can nest subroutines up to four deep.
This concludes the section on basic program commands. You can write some awesome programs using just what was described here. But, there is much more that the Stamp can do and we urge the advanced Stamper to read through the complete BASIC Stamp Manual which describes the entire PBASIC programming language.

**Coding style**

Style refers to your own particular style for creating code and includes layout, conventions for using case, headers, and use of comments. All code must follow correct syntax, but there are many different styles you can use. Here are some suggestions:

- Start every program with a comment header that has the program name and perhaps a brief description of what the program does.

- Use indentation to line things up. Labels in the first column, then use indents in multiples of 2 or 4 to mark code chunks, things inside loops and so on.

- Mark major sections or functions with a comment header line or two

- Have just the right number of comments, not too few and not too many. Assume the reader knows the programming language so have the comment be instructive. Here is an example of an instructive comment

  ```
  high 0  ' turn on motor
  ```

  and here is a useless comment

  ```
  high 0  ' set pin 0 high
  ```

  You need not comment every line. In fact, commenting every line is generally bad practice. Finally, add the comments when you create the code. If you tell yourself, "Oh, I'll add the comments when the code is finished", you'll never do it.

### 6. PBASIC language reference (abbreviated)

Here is a summary of the most useful PBASIC commands including the ones listed in the previous section plus a few others. For more information on these commands, consult the BASIC Stamp manual.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description [example]</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>Display info on host PC [debug dec x] [debug &quot;hello&quot;]</td>
</tr>
<tr>
<td>end</td>
<td>Stop program execution. Used at end of program</td>
</tr>
<tr>
<td>for...next</td>
<td>Loop. For marks start of loop, next marks end [for j = 1 to 3]</td>
</tr>
<tr>
<td>freqout</td>
<td>Use to generate sounds</td>
</tr>
<tr>
<td>gosub addr</td>
<td>Jump to subroutine marked by label addr [gosub alert]</td>
</tr>
<tr>
<td>goto addr</td>
<td>Jump to program line marked by label addr [goto start]</td>
</tr>
<tr>
<td>high n</td>
<td>Set pin n to +5V where n = 0 to 15 [high 3]</td>
</tr>
<tr>
<td>if cond then addr</td>
<td>Conditional branch [here; if in4 = 1 then here]</td>
</tr>
<tr>
<td><strong>input n</strong></td>
<td>Configure pin ( n ) to act as an input [input 2]</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><strong>low n</strong></td>
<td>Set pin ( n ) to 0V where ( n = 0 ) to 15 [low 3]</td>
</tr>
<tr>
<td><strong>output n</strong></td>
<td>Configure pin ( n ) to act as an output [output 1]</td>
</tr>
<tr>
<td><strong>pause x</strong></td>
<td>Pause program for ( x ) millisecond, ( x = 0 ) to 65,535 [pause 1000]</td>
</tr>
<tr>
<td><strong>pwm</strong></td>
<td>Pulse-width modulation on a pin to generate analog outputs</td>
</tr>
<tr>
<td><strong>random var</strong></td>
<td>Random number generator, leaves result in word variable ( var )</td>
</tr>
<tr>
<td><strong>rctime</strong></td>
<td>Use to measure value of potentiometers</td>
</tr>
<tr>
<td><strong>return</strong></td>
<td>Return from subroutine</td>
</tr>
<tr>
<td><strong>serin</strong></td>
<td>Read in serial data</td>
</tr>
<tr>
<td><strong>serout</strong></td>
<td>Send out serial data</td>
</tr>
<tr>
<td><strong>toggle n</strong></td>
<td>Toggle state of output pin ( n ) where ( n = 0 ) to 15 [toggle 3]</td>
</tr>
</tbody>
</table>
7. Controlling a small DC motor

The Stamp can control a small DC motor through a transistor switch. You will need a TIP120 transistor, a 1K resistor a 9V battery with battery snap and a motor.

The TIP120 pins look like this and on a schematic the pins are like this

Here is the schematic diagram for how to connect the motor

And here is a pictorial diagram for how to connect the components. The connections can be soldered or they can be made through the white breadboard prototyping area on the Stamp board or they can be made through an external breadboard.
PIN2 can be any I/O pin on your Stamp, for example. Connect the minus of the battery to the emitter of the transistor (E pin) and also connect the emitter of the transistor to ground (Vss) on the Stamp board.

To check if things are working, take a jumper wire and short the collector to the emitter pins of the transistor. The motor should turn on. Now, with the Stamp 9V battery in place, disconnect the 1K resistor from P2 and jumper over to +5V (Vdd). The motor should turn on. Put the resistor back into P2 and run the following test program:

```
high 2 : pause 1000 : low 2 : end
```

The motor should turn on for 1 second.
8. Resources

*The BASIC Stamp Manual*, available for free from the Parallax Web site

*BASIC Stamp Application Notes*, available for free from the Parallax Web site

*What's a Microcontroller*, available for free from the Parallax Web site


*The Microcontroller Application Cookbook* by Matt Gilliland

*The Nuts and Volts of BASIC Stamps*, vols 1 & 2, by Edwards, Glazner and Williams

*The Robot Builder's Bonanza* by Gordon McComb

*Practical Electronics for Inventors* by Paul Scherz

*Practical Robotics* by Bill Davies

*Robotic Explorations* by Fred Martin

*Robot Building for Beginners* by David Cook

*Mobile Robots* by Jones, Seiger and Flynn

*Build Your Own Robot* by Karl Lunt

*The Art of Electronics* by Horowitz and Hill

Please send comments, suggestions and typo alerts about this guide to wkdurfee@umn.edu