BRANCH

BRANCH offset,(address0,address1,...addressN)

Go to the address specified by offset (if in range).

- **Offset** is a variable/constant that specifies the address to branch to (0–N).
- **Addresses** are labels that specify where to branch.

Branch works like the ON x GOTO command found in other BASICs. It's useful when you want to write something like this:

```
if b2 = 0 then case_0   ' b2=0: go to label "case_0"
if b2 = 1 then case_1   ' b2=1: go to label "case_1"
if b2 = 2 then case_2   ' b2=2: go to label "case_2"
```

You can use Branch to organize this into a single statement:

```
BRANCH b2,(case_0,case_1,case_2)
```

This works exactly the same as the previous IF...THEN example. If the value isn't in range (in this case if b2 is greater than 2), Branch does nothing. The program continues with the next instruction after Branch.

Branch can be teamed with the Lookdown instruction to create a simplified SELECT CASE statement. See Lookdown for an example.

Sample Program:

```
Get_code:
        serin 0,N2400,("code"),b2              ' Get serial input.
                                               ' Wait for the string "code",
                                               ' then put next value into b2.
        BRANCH b2,(case_0,case_1,case_2)       ' If b2=0 then case_0
                                               ' If b2=1 then case_1
                                               ' If b2=2 then case_2
goto Get_code                                  ' If b2>2 then Get_code.

        case_0:          ...                   ' Destinations of the
        case_1:          ...                   ' Branch instruction.
        case_2:          ...
```

BUTTON

BUTTON pin,downstate,delay,rate,bytevariable,targetstate,address

Debounce button input, perform auto-repeat, and branch to address if button is in target state. Button circuits may be active-low or active-high (see the diagram on the next page).

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **Downstate** is a variable/constant (0 or 1) that specifies which logical state is read when the button is pressed.
- **Delay** is a variable/constant (0–255) that specifies how long the button must be pressed before auto-repeat starts. The delay is measured in cycles of the Button routine. Delay has two special settings: 0 and 255. If set to 0, the routine returns the button state with no debounce or auto-repeat. If set to 255, the routine performs debounce, but no auto-repeat.
- **Rate** is a variable/constant (0–255) that specifies the auto-repeat rate. The rate is expressed in cycles of the Button routine.
- **Bytevariable** is the workspace for Button. It must be cleared to 0 before being used by Button for the first time.
- **Targetstate** is a variable/constant (0 or 1) that specifies which state the button should be in for a branch to occur (0=not pressed, 1=pressed).
- **Address** is a label that specifies where to branch if the button is in the target state.

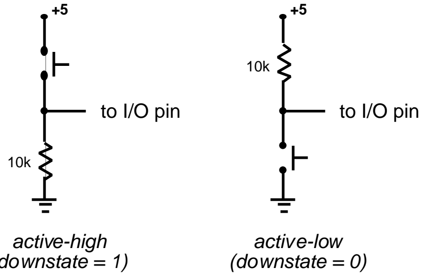
When you press a button or flip a switch, the contacts make or break a connection. A burst of electrical noise occurs as the contacts bounce against each other. Button's debounce feature prevents this noise from being interpreted as more than one switch action.

Button also lets the Stamp react to a button press the way your PC keyboard does to a key press. When you press a key, a character appears on the screen. If you hold the key down, there's a delay, then a rapid-fire stream of characters appears on the screen. Button's autorepeat function can be set up to work the same way.

Button is designed to be used inside a program loop. Each time through the loop, Button checks the state of the specified pin. When

BUTTON *(continued)*

it first matches *downstate*, Button debounces the switch. Then, in accordance with *targetstate*, it either branches to *address* (targetstate = 1) or doesn't (targetstate = 0).



Example button circuits.

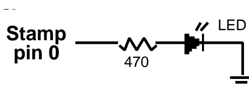
If the switch is kept in *downstate*, Button tracks the number of program loops that execute. When this count equals *delay*, Button again triggers the action specifed by *targetstate* and *address*. Hereafter, if the switch remains in *downstate*, Button waits *rate* number of cycles between actions.

The important thing to remember about Button is that it does not stop program execution. In order for its delay and autorepeat functions to work, Button must execute from within a loop.

Sample Program:

```
' This program toggles (inverts) the state of an LED on pin 0 when the
' active-low switch on pin 7 is pressed. When the switch is held down, Button
' waits, then rapidly autorepeats the Toggle instruction, making the LED
' flash rapidly. When the switch is not pressed, Button skips the Toggle
' instruction. Note that b2, the workspace variable for Button, is cleared
' before its first use. Don't clear it within the loop.

        let b2 = 0                              ' Button workspace cleared.
Loop:
        BUTTON 7,0,200,100,b2,0,skip            ' Go to skip unless pin7=0.
        Toggle 0                                ' Invert LED.
        ...                                     ' Other instructions.
skip:
        goto Loop                               ' Skip toggle and go to Loop.
```



LED hookup for sample program.

DEBUG

DEBUG variable{,variable}

Displays the specified variable (bit, byte, or word) in a window on the screen of a connected PC. Debug works only after a "run" (ALT-R) download has finished.

Debug accepts formatting modifiers as follows:

- No modifiers: prints "variable = value"
- # before variable, as in #b2, prints the decimal value, without the "variable =" text.
- $ before variable, as in $b2, prints hex value.
- % before variable, as in %b2, prints binary value.
- @ before variable, as in @b2, prints the ASCII character corresponding to the value of the variable.
- Text in quotes appears as typed.
- cr (carriage return) causes printing in the Debug window to start a new line.
- cls (clear screen) clears the Debug window.
- commas must separate all variables used with Debug.

Samples:

```
DEBUG b2                        ' Print "b2 = " + value of b2
DEBUG #b2                       ' Print value of b2
DEBUG "reading is ",b2          ' Print "reading is " & value of b2
DEBUG #%b2                      ' Print value of b2 in binary
DEBUG #@b2                      ' Display the ASCII character
                                ' corresponding to the value in b2.
DEBUG "inputs ",b2,b3,cr        ' Print "inputs" & value of b2 & value
                                ' of b3 & carriage return.
```

EEPROM

EEPROM {location},(data,data,...)

Store values in EEPROM before downloading the BASIC program.

- **Location** is an optional variable/constant (0–255) that specifies the starting location in the EEPROM at which data should be stored. If no location is given, data is written starting at the next available location.
- **Data** are variables/constants (0–255) to be stored sequentially in the EEPROM.

EEPROM is useful for storing values to be used by your program. One application is to store long messages for use by Serout as shown below:

Program Sample 1:

```
' Sends the text "A very long message indeed..." then reads address 255 for
' the last instruction location of the program.
        serout 0,N2400,("A very long message indeed...")
        read 255,b2             ' Get last program location (reflects length of program)
        debug b2                ' Display it on the screen.
```

Program Sample 2:

```
' Sends the text "A very long message indeed..." then reads address 255 for
' the last instruction location of the program.
        EEPROM 0,("A very long message indeed...")
        for b2 = 0 to 28        ' Send message 1 char at a time.
        read b2,b3              ' Read data at location b2 of
        serout 0,N2400,(b3)     ' EEPROM into b3. Transmit b3.
        next                    ' Send next character.
        read 255,b2             ' Get last program location (reflects length of program)
        debug b2                ' Display it on the screen.
```

The first program sample shows an endpoint of 197, meaning that it uses 58 bytes of program memory to send the 29-byte message. Sample 2 has an endpoint of 232 (23 bytes of program memory used). When you add 29 bytes for the storage of the message, sample 2 is 6 bytes more efficient. The savings are greater when the messages are used at several points in a program.

END

END

Enter sleep mode indefinitely. The Stamp wakes up when the power cycles or the PC connects. Power consumption is reduced to about 20 μA, assuming no loads are being driven.

If you do leave Stamp pins in an output-high or output-low state driving loads when End executes, two things will happen:

- The loads will continue to draw current from the Stamp's power supply.
- Every 2.3 seconds, current to those loads will be interrupted for a period of approximately 18 milliseconds (ms).

The reason for the output glitch every 2.3 seconds has to do with the design of the PBASIC interpreter chip. It has a free-running clock called the "watchdog timer" that can periodically reset the processor, causing a sleeping Stamp to wake up. Once awake, the Stamp checks its program to determine whether it should remain awake or go back to sleep. After an End instruction, the Stamp has standing orders to go back to sleep.

Unfortunately, the watchdog timer cannot be shut off, so the Stamp actually gets its sleep as a series of 2.3-second naps. At the end of each nap, the watchdog timer resets the PBASIC chip. Among other things, a reset causes all of the chip's pins to go into input mode. It takes approximately 18 ms for the PBASIC firmware to get control, restore the pins to their former state, and put the Stamp back to sleep.

If you use End, Nap, or Sleep in your programs, make sure that your loads can tolerate these periodic power outages. The easy solution is often to connect pull-up or pull-down resistors as appropriate to ensure a continuing supply of current during the reset glitch.

FOR...NEXT

FOR variable = start TO end {STEP {-} increment}...NEXT {variable}

Establish a For...Next loop. *Variable* is set to the value *start*. Code between the For and Next instructions is then executed. *Variable* is then incremented/decremented by *increment* (if no increment value is given, the variable is incremented by 1). If *variable* has not reached or passed the value *end*, the instructions between For and Next are executed again. If *variable* has reached or passed *end*, then the program continues with the instruction after Next. The loop is executed at least once, no matter what values are given for *start* and *end*.

Your program can have as many For...Next loops as necessary, but they cannot be nested more than eight deep (in other words, your program can't have more than eight loops within loops).

- **Variable** is a bit, byte, or word variable used as an internal counter. *Start* and *end* are limited by the capacity of *variable* (bit variables can count from 0 to 1, byte variables from 0 to 255, and word variables from 0 to 65535).
- **Start** is a variable/constant which specifies the initial value of *variable*.
- **End** is a variable/constant which specifies the ending value of *variable*.
- **Increment** is an optional variable/constant by which the counter increments or decrements (if negative). If no step value is given, the variable increments by 1.
- **Variable** (after Next) is optional. It is used to clarify which of a series of For...Next loops a particular Next refers to.

Program Samples:

Programmers most often use For...Next loops to repeat an action a fixed number of times, like this:

```
FOR b2 = 1 to 10                  ' Repeat 10 times.
 debug b2                         ' Show b2 in debug window.
NEXT                              ' Again until b2>10.
```

FOR...NEXT *(continued)*

Don't overlook the fact that all of the parameters of a For...Next loop can be variables. Not only can your program establish these values itself, it can also modify them while the loop is running. Here's an example in which the step value increases with each loop:

```
let b3 = 1
FOR b2 = 1 to 100 STEP b3          ' Each loop, add b3 to b2.
debug b2                           ' Show b2 in debug window.
let b3 = b3+2                      ' Increment b3.
NEXT                               ' Again until b2>15.
```

If you run this program, you may notice something familiar about the numbers in the debug window (1,4,9,16,25,36,49...). They are all squares (1^2=1, 2^2=4, 3^2=9, 4^2=16, etc.), but our program used addition, not multiplication, to calculate them. This method of generating a polynomial function is credited to Sir Isaac Newton.

There is a potential bug in the For...Next structure. PBASIC uses 16-bit integer math to increment/decrement the counter variable and compare it to the *end* value. The maximum value a 16-bit variable can hold is 65535. If you add 1 to 65535, you get 0 (the 16-bit register rolls over, much like a car's odometer does when you exceed the maximum mileage it can display).

If you write a For...Next loop whose *step* value is larger than the difference between the *end* value and 65535, this rollover will cause the loop to execute more times than you expect. Try the following:

```
FOR w1 = 0 to 65500 STEP 3000      ' Each loop add 3000 to w1.
  debug w1                         ' Show w1 in debug window.
NEXT                               ' Again until w1>65500.
```

The value of w1 increases by 3000 each trip through the loop. As it approaches the stop value, an interesting thing happens: 57000, 60000, 63000, 464, 3464... It passes the *end* value and keeps going. That's because the result of the calculation 63000 + 3000 exceeds the maximum capacity of a 16-bit number. When the value rolls over to 464, it passes the test "is w1 > 65500?" used by Next to determine when to end the loop.

The same problem can occur when the step value is negative and larger (in absolute value) than the difference between the *end* value and 0.

GOSUB

GOSUB address

Store the address of the instruction following Gosub, branch to *address*, and continue execution there. The next Return instruction takes the program back to the stored address, continuing the program on the instruction following the most recent Gosub.

- **Address** is a label that specifies where to branch. Up to 16 GOSUBs are allowed per program.

PBASIC stores data related to Gosubs in variable w6. Make sure that your program does not write to w6 unless all Gosubs have Returned, and don't expect data written to w6 to be intact after a Gosub.

If a series of instructions is used at more than one point in your program, you can turn those instructions into a subroutine. Then, wherever you would have inserted that code, you can simply write Gosub *label* (where *label* is the name of your subroutine).

Sample Program:

```
' In this program, the subroutine test takes a pot measurement, then performs
' a weighted average by adding 1/4 of the current measurement to 3/4 of a
' previous measurement. This has the effect of smoothing out noise.
        for b0 = 1 to 10
        GOSUB test              ' Save this address & go to test.
        serout 1,N2400,(b2)     ' Return here after test.
        next                    ' Again until b0 > 10.
        end                     ' Prevents program from running into test.
test:
        pot 0,100,b2            ' Take a pot reading.
        let b2 = b2/4 + b4      ' Make b2 = (.25*b2)+b4.
        let b4 = b2 * 3 / 4     ' Make b4 = .75*b2.
return                          ' Return to previous address+1 instruction.
```

The Return instruction at the end of *test* sends the program to the instruction immediately following Gosub *test*; in this case Serout.

Make sure that your program cannot wander into a subroutine without Gosub. In the sample, what if End were removed? After the loop , execution would continue in *test*. When it reached Return, the program would jump back into the the For...Next loop at Serout because this was the last return address assigned. The For...Next loop would execute forever.

GOTO

GOTO address

Branch to *address*, at which point execution continues.

- **Address** is a label that specifies where to branch.

Sample Program:

```
abc:
        pulsout 0,100           ' Generate a 1000-µs pulse on pin 0.
        GOTO abc                ' Repeat forever.
```

HIGH

HIGH pin

Make the specified pin output high. If the pin is programmed as an input, it changes to an output.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin.

You can think of the High instruction as the equivalent of:

```
output 3            ' Make pin 3 an output.
let pin3 = 1        ' Set pin 3 high.
```

Notice that the Output command accepts the pin number (3), while Let requires the pin's variable name *pin3*. So, in addition to saving one instruction, High allows you to make a pin output-high using only its number. When you look at the sample program below, imagine how difficult it would be to write it using Output and Let.

This points out a common bug involving High. Programmers sometimes substitute pin names like *pin3* for the pin number. Remember that those pin names are really bit variables. As bits, they can hold values of 0 or 1. The statement "High pin3" is a valid BASIC instruction, but it means, "Get the state of *pin3*. If *pin3* is 0, make *pin 0* output high. If *pin3* is 1, make *pin 1* output high."

Sample Program:

```
' One at a time, change each of the pins to output and set it high.
    for b2 = 0 to 7        ' Eight pins (0-7).
    HIGH b2                ' Set pin no. indicated by b2.
    pause 500              ' Wait 1/2 second between pins.
    next                   ' Do the next pin.
```

IF...THEN

IF variable ?? value {AND/OR variable ?? value...} THEN address

Compare variable(s) to value(s) and branch if result is true.

- **??** is one of the following operators: = (equal), <> (not equal), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to)
- **Variable** is a variable that is compared to value(s)
- **Value** is a variable or constant for comparison
- **Address** is a label that specifies where to branch if the result of the comparison(s) is true

Unlike those in some other flavors of BASIC, this If...Then statement can only go to an address label. It does not support statements like “IF x > 30 THEN x = 0.” To do the same thing neatly in PBASIC requires a little backwards thinking:

```
IF x <= 30 THEN skip                    ' If x is less than or equal
let x = 0                               ' to 30, don't make x=0.
skip:...                                ' Program continues.
```

Unless x > 30, the program skips over the instruction “let x = 0.”

PBASIC’s If...Then can evaluate two or more comparisons at one time with the conjunctions *And* and *Or*. It works from left to right, and does not accept parentheses to change the order of evaluation. It can be tricky to anticipate the outcome of compound comparisons. We suggest that you set up a test of your logic using debug as shown in the sample program below.

Sample Program:

```
' Evaluates the If...Then statement and displays the result in a debug window.
    let b2 = 7                              ' Assign values.
    let b3 = 16
    IF b3 < b2 OR b2 = 7 THEN True          ' B3 is not less than b2, but
                                            ' b2 is 7: so statement is true.
    debug "statment is false"               ' If statement is false, goto here.
end
True:
    debug "statement is true"               ' If statement is true, goto here.
end
```

INPUT

INPUT pin

Make the specified pin an input. This turns off the pin's output drivers, allowing your program to read whatever state is present on the pin from the outside world.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.

There are several ways to set pins to input. When a program begins, all of the Stamp's pins are inputs. Input instructions (Pulsin, Serin) automatically change the specified pin to input and leave it in that state. Writing 0s to particular bits of the variable *dirs* makes the corresponding pins inputs. And then there's the Input instruction.

When a pin is set to input, your program can check its state by reading its value. For example:

```
Hold:    if pin4 = 0 then Hold          ' Stay here until pin4 is 1.
```

The program is reading the state of pin 4 as set by external circuitry. If nothing is connected to pin 4, it could be in either state (1 or 0) and could change states apparently at random.

What happens if your program writes to a pin that is set up as an input? The state is written to the output latch assigned to the pin. Since the output drivers are disconnected when a pin is an input, this has no effect. If the pin is changed to output, the last value written to the latch will appear on the pin. The program below shows how this works.

Sample Program:

```
' To see this program in action, connect a 10k resistor from pin 7 to +5V.
' When the program runs, a debug window will show you the state at pin 7
' (a 1, due to the +5 connection); the effect of writing to an input pin (none);
'  and the result of changing an input pin to output (the latched state appears
' on the pin and may be read by your program). Finally, the program shows
' how changing pin 7 to output writes a 1 to the corresponding bit of dirs.
    INPUT 7                                  ' Make pin 7 an input.
    debug "State present at pin 7: ",#pin7,cr,cr
    let pin7 = 0                             ' Write 0 to output latch.
    debug  "After 0 written to input: ",#pin7,cr,cr
    output 7                                 ' Make pin 7 an output.
    debug  "After pin 7 changed to output: ",#pin7,cr
    debug "Dirs (binary): ",#%dirs           ' Show contents of dirs.
```

LET

{LET} variable = {-}value ?? value…

Assign a value to the variable and/or perform logic operations on the variable. All math and logic is done at the word level (16 bits).

The instruction “Let” is optional. For instance, “A=10” is identical to “Let A=10”.

- **??** is one of the following operators:

```
+       add
–       subtract
*       multiply (returns low word of result)
**      multiply (returns high word of result)
/       divide (returns quotient)
//      divide (returns remainder)
min     keep variable greater than or equal to value
max     keep variable less than or equal to value
&       logical AND
|       logical OR
^       logical XOR
&/      logical AND NOT
|/      logical OR NOT
^/      logical XOR NOT
```

- **Variable** is assigned a value and/or manipulated.
- **Value(s)** is a variable/constant which affects the variable.

When you write programs that perform math, bear in mind the limitations of PBASIC’s variables: all are positive integers; bits can represent 0 or 1; bytes, 0 to 255; and words, 0 to 65535. PBASIC doesn’t understand floating-point numbers (like 3.14), negative numbers (–73), or numbers larger than 65535.

In most control applications, these are not serious limitations. For example, suppose you needed to measure temperatures from -50° to +200°F. By assigning a value of 0 to –50° and 65535 to +200° you would have a resolution of 0.0038°!

The integer restriction doesn’t mean you can’t do advanced math with the Stamp. You just have to improvise . Suppose you needed to use the constant (3.14159...) in a program. You would like to write:

```
Let w0 = b2 * 3.14
```

LET *(continued)*

However, the number "3.14" is a floating-point number, which the Stamp doesn't understand. There is an alternative. You can express such quantities as fractions. Take the value 1.5. It is equivalent to the fraction 3/2. With a little effort you can find fractional substitutes for most floating-point values. For instance, it turns out that the fraction 22/7 comes very close to the value of . To perform the calculation *Let w0 = b2 * 3.14*, the following instruction will do the trick:

```
Let w0 = b2 * 22 / 7
```

PBASIC works out problems from left to right. You cannot use parentheses to alter this order as you can in some other BASICs. And there is no "precedence of operators" that (for instance) causes multiplication to be done before addition. Many BASICs would evaluate the expression "2+3*4" as 14, because they would calculate "3*4" first, then add 2. PBASIC, working from left to right, evaluates the expression as 20, since it calculates "2+3" and multiplies the result by 4. When in doubt, work up an example problem and use debug to show you the result.

Sample Program:

```
pot 0,100,b3          ' Read pot, store result in b3.
LET b3=b3/2           ' Divide result by 2.
b3=b3 max 100         ' Limit result to 0-100.
                      ' Note that "LET" is not necessary.
```

LOOKDOWN

LOOKDOWN target,(value0,value1,...valueN),variable

Search *value(s)* for *target* value. If target matches one of the values, store the matching value's position (0–N) in *variable*.

If no match is found, then the variable is unaffected.

- **Target** is the variable/constant being sought.
- **Value0, value1,...** is a list of values. The target value is compared to these values
- **Variable** holds the result of the search.

Lookdown's ability to convert an arbitrary sequence of values into an orderly sequence (0,1,2...) makes it a perfect partner for Branch. Using Lookdown and Branch together, you can create a SELECT CASE statement.

Sample Program:

```
' Program receives the following one-letter instructions over a serial
' linkand takes action: (G)o, (S)top, (L)ow, (M)edium, (H)igh.
Get_cmd: serin 0,N2400,b2' Put input value into b2.
        LOOKDOWN b2,("GSLMH"),b2            ' If b2="G" then b2=0 (see note)
                                            ' If b2="S" then b2=1
                                            ' If b2="L" then b2=2
                                            ' If b2="M" then b2=3
                                            ' If b2="H" then b2=4
        branch b2,(go,stop,low,med,hi)      ' If b2=0 then go
                                            ' If b2=1 then stop
                                            ' If b2=2 then low
                                            ' If b2=3 then med
                                            ' If b2=3 then hi
goto Get_cmd                                ' Not in range; try again.
go:     ...                     ' Destinations of the
stop: ...                       ' Branch instruction.
low:  ...
med:...
hi:           ...
' Note: In PBASIC instructions, including EEPROM, Serout, Lookup and
' Lookdown, strings may be formatted several ways. The Lookdown command
' above could also have been written:
'       LOOKDOWN b2,(71,83,76,77,72),b2     ' ASCII codes for "G","S","L"...
' or
'       LOOKDOWN b2,("G","S","L","M","H"),b2
```

LOOKUP

LOOKUP offset,(value0,value1,...valueN),variable

Look up data specified by *offset* and store it in *variable*. For instance, if the values were 2, 13, 15, 28, 8 and *offset* was 1, then *variable* would get the value "13", since "13" is the second value in the list (the first value is #0, the second is #1, etc.). If *offset* is beyond the number of values given, then *variable* is unaffected.

- **Offset** specifies the index number of the value to be looked up.
- **Value0, value1,...** is a table of values.
- **Variable** holds the result of the lookup.

Many applications require the computer to calculate an output value based on an input value. When the relationship is simple, like "out = 2*in", it's no problem at all. But what about relationships that are not so obvious? In PBASIC you can use Lookup.

For example, stepper motors work in an odd way. They require a changing pattern of 1s and 0s controlling current to four coils. The sequence appears in the table to the right.

Step #	Binary	Decimal
0	1010	10
1	1001	9
2	0101	5
3	0110	6

Repeating that sequence makes the motor turn. The program below shows how to use a Lookup table to generate the sequence.

Sample Program:

```
' Output the four-step sequence to drive a stepper motor w/on-screen simulation.
        let dirs = %00001111                    ' Set lower 4 pins to output.
Rotate:
        for b2 = 0 to 3
          LOOKUP b2,(10,9,5,6),b3               ' Convert offset (0-3)
                                                ' to corresponding step.
          let pins = b3                         ' Output the step pattern.
          LOOKUP b2,("|/-\"),b3                 ' Convert offset (0-3)
                                                ' to "picture" for debug.
          debug cls,#%pins,"  ",#@b3            ' Display value on pins,
        next                                    ' animated "motor."
goto Rotate                                     ' Do it again.

' Note: In the debug line above, there are two spaces between the quotes.
```

LOW

LOW pin

Make the specified pin output low. If the pin is programmed as an input, it changes to an output.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.

You can think of the Low instruction as the equivalent of:

```
output 3          ' Make pin 3 an output.
let pin3 = 0      ' Make pin 3 low.
```

Notice that the Output command accepts the pin number (3), while Let requires the pin's variable name *pin3*. So, in addition to saving one instruction, Low allows you to make a pin output-low using only its number. When you look at the sample program below, imagine how difficult it would be to write it using Output and Let.

This also points out a common bug involving Low. Programmers sometimes substitute pin names like *pin3* for the pin number. Remember that those pin names are really bit variables. As bits, they can hold values of 0 or 1. The statement "Low pin3" is a valid PBASIC instruction, but it means, "Get the state of *pin3*. If *pin3* is 0, make *pin 0* output low. If *pin3* is 1, make *pin 1* output low."

Sample Program:

```
' One at a time, change each of the pins to output and make it low.
    for b2 = 0 to 7       ' Eight pins (0-7).
    LOW b2                ' Clear pin no. indicated by b2.
    pause 500             ' Wait 1/2 second between pins.
    next                  ' Do the next pin.
```

NAP

NAP period

Enter sleep mode for a short period. Power consumption is reduced to about 20 μA, assuming no loads are beingdriven.

- **Period** is a variable/constant which determines the duration of the reduced power nap. The duration is (2^period) * approximately 18 ms. *Period* can range from 0 to 7, resulting in the nap lengths shown in the table.

Period	2^{period}	Nap Length
0	1	18 ms
1	2	36 ms
2	4	72 ms
3	8	144 ms
4	16	288 ms
5	32	576 ms
6	64	1152 ms
7	128	2304 ms

Nap uses the same shutdown/startup mechanism as Sleep, with one big difference. During sleep, the Stamp compensates for variations in the speed of the watchdog timer that serves as its alarm clock. As a result, longer sleep intervals are accurate to about ±1 percent. Naps are controlled by the watchdog timer without compensation. Variations in temperature, voltage, and manufacturing of the PBASIC chip can cause the actual timing to vary by as much as –50, +100 percent (i.e., a period-0 nap can range from 9 to 36 ms).

If your Stamp application is driving loads (sourcing or sinking current through output-high or output-low pins) during a nap, current will be interrupted for about 18 ms when the Stamp wakes up. The reason is that the reset that awakens the Stamp also switches all of the pins input mode for about 18 ms. When PBASIC regains control, it restores the I/O direction dictated by your program.

When you use End, Nap, or Sleep, make sure that your loads can tolerate these glitches. The simplest way is often to connect resistors high or low (to +5V or ground) as appropriate to ensure a continuing supply of current during reset.

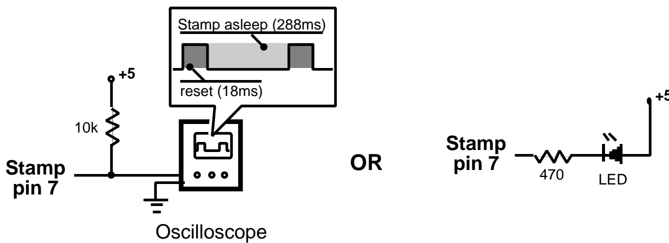
The sample program on the next page can be used to demonstrate the effects of the nap glitch with either an LED and resistor, or an oscilloscope, as shown in the diagram.

NAP *(continued)*

Sample Program:

```
' During the Nap period, the Stamp will continue to drive loads connected to
' pins that are configured as outputs. However, at the end of a Nap, all pins
' briefly change to input, interrupting the current. This program may be
' used to demonstrate the effect.

        low 7                   ' Make pin 7 output-low.
Again:
        NAP 4                   ' Put the Stamp to sleep for 288 ms.
goto Again                      ' Nap some more.
```



Use either of these circuits to observe the output glitch when the Stamp awakens from a Nap. Pin 7 is output low while the Stamp is asleep. When it resets, all pins switch to input, allowing the resistor to pull pin 7 high (left) or causing the LED to blink off (right).

OUTPUT

OUTPUT pin

Make the specified pin an output.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.

When a program begins, all of the Stamp's pins are inputs. If you want to drive a load, like an LED or logic circuit, you must configure the appropriate pin as an output.

Output instructions (High, Low, Pulsout, Serout, Sound and Toggle) automatically change the specified pin to output and leave it in that state. Although not technically an output instruction, Pot also changes a pin to output. Writing 1s to particular bits of the variable *Dirs* causes the corresponding pins to become outputs. And then there's Output.

When a pin is configured as an output, you can change its state by writing a value to it, or to the variable *Pins*. When a pin is changed to output, it may be a 1 or a 0, depending on values previously written to the pin. To guarantee which state a pin will be in, either use the High or Low instructions to change it to output, or write the appropriate value to it immediately before switching to output.

Sample Program:

```
' To see this program in action, connect a 10k resistor from pin 7 to the +5
' power-supply rail. When the program runs, a debug window will show you the
' the state at pin 7 (a 1, due to the +5 connection); the effect of writing
' to an input pin (none); and the result of changing an input pin to output
' (the latched state appears on the pin and may be read by your program).
' Finally, the program will show how changing pin 7 to output wrote
' a 1 to the corresponding bit of the variable Dirs.

    input 7                                         ' Make pin 7 an input.
    debug "State present at pin 7: ",#pin7,cr,cr
    let pin7 = 0                                    ' Write 0 to output latch.
    debug  "After 0 written to input: ",#pin7,cr,cr
    OUTPUT 7                                        ' Make pin 7 an output.
    debug  "After pin 7 changed to output: ",#pin7,cr
    debug "Dirs (binary): ",#%dirs
```

PAUSE

PAUSE milliseconds

Pause program execution for the specified number of milliseconds.

- **Milliseconds** is a variable/constant (0–65535) that specifies how many milliseconds to pause.

The delays produced by the Pause instruction are as accurate as the Stamp's ceramic resonator timebase, ±1 percent. When you use Pause in timing-critical applications, keep in mind the relatively low speed of the BASIC interpreter (about 2000 instructions per second). This is the time required for the PBASIC chip to read and interpret an instruction stored in the EEPROM.

Since the PBASIC chip takes 0.5 milliseconds to read in the Pause instruction, and 0.5 milliseconds to read in the instruction following it, you can count on loops involving Pause taking at least 1 millisecond longer than the Pause period itself. If you're programming timing loops of fairly long duration, keep this (and the 1-percent tolerance of the timebase) in mind.

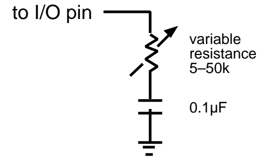
Sample Program:

```
abc:
        low 2                   ' Make pin 2 output low.
        PAUSE 100               ' Pause for 0.1 second.
        high 2                  ' Make pin 2 output high.
        PAUSE 100               ' Pause for 0.1 second.
goto abc
```

POT

POT pin,scale,variable

Read a 5–50k potentiometer, thermistor, photocell, or other variable resistance. The pin specified by Pot must be connected to one side of a resistor, whose other side is connected through a capacitor to ground. A resistance measurement is taken by timing how long it takes to discharge the capacitor through the resistor. If the pin is an input when Pot executes, it will be changed to output and left in the output-high state.



- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **Scale** is a variable/constant (0–255) used to scale the instruction's internal 16-bit result. The 16- bit reading is multiplied by (scale/ 256), so a scale value of 128 would reduce the range by approximately 50%, a scale of 64 would reduce to 25%, and so on. The Alt-P option (explained below) provides a means to find the best scale value for a particular resistor.
- **Variable** is used to store the final result of the reading. Internally, the Pot instruction calculates a 16-bit value, which is scaled down to an 8-bit value. The amount by which the internal value must be scaled varies with the size of the resistor being used.

Finding the best Pot scale value:

- To find the best scale value, connect the resistor to be used with the Pot instruction to the Stamp, and connect the Stamp to the PC.
- Press Alt-P while running the Stamp's editor software. A special calibration window appears, allowing you to find the best value.
- The window asks for the number of the I/O pin to which the resistor is connected. Select the appropriate pin (0-7).
- The editor downloads a short program to the Stamp (this overwrites any program already stored in the Stamp).

POT *(continued)*

- Another window appears, showing two numbers: scale and value. Adjust the resistor until the smallest possible number is shown for scale (we're assuming you can easily adjust the resistor, as with a potentiometer).

Once you've found the smallest number for scale, you're done. This number should be used for the scale in the Pot instruction.

Optionally, you can verify the scale number found above by pressing the spacebar. This locks the scale and causes the Stamp to read the resistor continuously. The window displays the value. If the scale is good, you should be able to adjust the resistor, achieving a 0–255 reading for the value (or as close as possible). To change the scale value and repeat this step, just press the spacebar. Continue this process until you find the best scale.

Sample Program:

```
abc:
        POT 0,100,b2                ' Read potentiometer on pin 0.
        serout 1,N300,(b2)          ' Send potentiometer reading
                                    ' over serial output.
goto abc                            ' Repeat the process.
```

PULSIN

PULSIN pin,state,variable

Change the specified pin to input and measure an input pulse in 10µs units.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **State** is a variable/constant (0 or 1) that specifies which edge must occur before beginning the measurement.
- **Variable** is a variable used to store the result of the measurement. The variable may be a word variable with a range of 1 to 65535, or a byte variable with a range of 1 to 255.

Many analog properties (voltage, resistance, capacitance, frequency, duty cycle) can be measured in terms of pulse durations. This makes Pulsin a valuable form of analog-to-digital conversion.

You can think of Pulsin as a fast stopwatch that is triggered by a change in state (0 or 1) on the specified pin. When the state on the pin changes to the state specified in Pulsin, the stopwatch starts counting. When the state on the pin changes again, the stopwatch stops.

If the state of the pin doesn't change (even if it is already in the state specified in the Pulsin instruction), the stopwatch won't trigger. Pulsin waits a maximum of 0.65535 seconds for a trigger, then returns with 0 in *variable*.

The variable can be either a word or a byte. If the variable is a word, the value returned by Pulsin can range from 1 to 65535 units of 10µs. If the variable is a byte, the value returned can range from 1 to 255 units of 10µs. Internally, Pulsin always uses a 16-bit timer. When your program specifies a byte, Pulsin stores the lower 8 bits of the internal counter into it. Pulse widths longer than 2550µs will give false, low readings with a byte variable. For example, a 2560 µs pulse returns a Pulsin reading of 256 with a word variable and 0 with a byte variable.

Sample Program:

```
PULSIN 4,0,w2          ' Measure an input pulse on pin 4.
serout 1,n300,(b5)     ' Send high byte of 16-bit pulse measurement
 . . .                 '  over serial output.
```

PULSOUT

PULSOUT pin,time

Generate a pulse by inverting a pin for a specified amount of time. If the pin is an input when Pulsout is executed, it will be changed to an output.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **Time** is a variable/constant (0–65535) that specifies the length of the pulse in 10µs units.

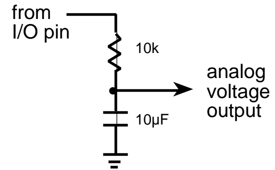
Sample Program:

```
abc:
        PULSOUT 0,3             ' Invert pin 0 for 30
                                ' microseconds.
        pause 1                 ' Pause for 1 ms.
goto abc                        ' Branch to abc.
```

PWM

PWM pin,duty,cycles

Output pulse-width-modulation on a pin, then return the pin to input state. PWM can be used to generate analog voltages (0-5V) through a pin connected to a resistor and capacitor to ground; the resistor-capacitor junction is the analog output (see circuit). Since the capacitor gradually discharges, PWM should be executed periodically to update and/or refresh the analog voltage.



- **Pin** is a variable/constant (0–7) which specifies the I/O pin to use.
- **Duty** is a variable/constant (0–255) which specifies the analog level desired (0–5 volts).
- **Cycles** is a variable/constant (0–255) which specifies the number of cycles to output. Larger capacitors require multiple cycles to fully charge. Each cycle takes about 5 ms.

PWM emits a burst of 1s and 0s whose ratio is proportional to the *duty* value you specify. If *duty* is 0, then the pin is continuously low (0); if *duty* is 255, then the pin is continuously high. For values in between, the proportion is *duty*/255. For example, if *duty* is 100, the ratio of 1s to 0s is 100/255 = 0.392, approximately 39 percent.

When such a burst is used to charge a capacitor arranged as shown in the schematic, the voltage across the capacitor is equal to (*duty*/255) * 5. So if *duty* is 100, the capacitor voltage is (100/255) * 5 = 1.96 volts.

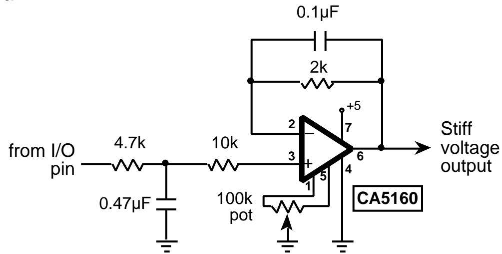
This voltage will drop as the capacitor discharges through whatever load it is driving. The rate of discharge is proportional to the current drawn by the load; more current = faster discharge. You can combat this effect in software by refreshing the capacitor's charge with frequent doses of PWM.

You can also buffer the output to greatly reduce the need for frequent PWM cycles. The schematic on the next page shows an example. Feel free to substitute more sophisticated circuits; this

PWM *(continued)*

"op-amp follower" is merely a suggestion.

If you use a buffer circuit, you will still have to refresh the capacitor from time to time. When the pin is configured to input after PWM executes, it is effectively disconnected from the resistor/capacitor circuit. However, leakage currents of up to 1µA can flow into or out of this "disconnected" pin. Over time, these small currents will cause the voltage on the capacitor to drift. The same applies for leakage current from the op-amp's input, as well as the capacitor's own internal leakage. Executing PWM occasionally will reset the capacitor voltage to the intended value.



Op-amp buffer for PWM.

One more thing: The name "PWM" may lead you to expect a neat train of fixed-width pulses for a given duty value. That's not the case. When viewed on an oscilloscope, the PWM output looks like a noisy jumble of varying pulsewidths. The only guarantee is that the overall ratio of highs to lows is in the proportion specified by *duty*.

Sample Program:

```
abc:
      serin 0,n300,b2          ' Receive serial byte.
      PWM 1,b2,20              ' Output an analog voltage proportional to
                               ' the serial byte received
```

RANDOM

RANDOM wordvariable

Generate the next pseudo-random number in *wordvariable*.

- **Wordvariable** is a variable (0–65535) that acts as the routine's workspace and its result. Each pass through Random leaves the next number in the pseudorandom sequence.

The Stamp uses a sequence of 65535 essentially random numbers to execute this instruction. When Random executes, the value in wordvariable determines where to "tap" into the sequence of random numbers. If the same initial value is always used in *wordvariable*, then the same sequence of numbers is generated. Although this method is not absolutely random, it's good enough for most applications.

To obtain truly random results, you must add an element of uncertainty to the process. For instance, your program might execute Random continuously while waiting for the user to press a button.

Sample Program:

```
loop:
    RANDOM w1           ' Generate a 16-bit random number.
    sound 1,(b2,10)     ' Generate a random tone on pin 1 using the low
                        ' byte of the random number b2 as the note number.
    goto loop           ' Repeat the process
```

READ

READ location,variable

Read EEPROM location and store value in *variable*.

- **Location** is a variable/constant (0–255) that specifies which location in the EEPROM to read from.
- **Variable** receives the value read from the EEPROM (0–255).

The EEPROM is used for both program storage (which builds downward from address 254) and data storage (which builds upward from address 0). To ensure that your program doesn't overwrite itself, read location 255 in the EEPROM before writing any data. Location 255 holds the address of the last instruction in your program. Therefore, your program can use any space below the address given in location 255. For example, if location 255 holds the value 100, then your program can use locations 0–99 for data.

Sample Program:

```
        READ 255,b2             ' Get location of last program instruction.
loop:
        b2 = b2 - 1             ' Decrement to next available EEPROM location
        serin 0,N300,b3         ' Receive serial byte in b3
        write b2,b3             ' Store received serial byte in next EEPROM location
if b2 > 0 then loop             ' Get another byte if there's room left to store it.
```

RETURN

RETURN

Return from subroutine. Return **branches back to the address following the most recent** Gosub **instruction, at which point program execution continues.**

Return **takes no parameters. For more information on using subroutines, see the** Gosub **listing.**

Sample Program:

```
for b4 = 0 to 10
gosub abc                   ' Save return address and then branch to abc.
next
abc:
      pulsout 0,b4          ' Output a pulse on pin 0.
                            ' Pulse length is b4 x 10 microseconds.
      toggle 1              ' Toggle pin 1.
      RETURN                ' Return to instruction following gosub.
```

REVERSE

REVERSE pin

Reverse the data direction of the given pin. If the pin is an input, make it an output; if it's an output, make it an input.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin.

See the Input and Output commands for more information on configuring pins' data directions.

Sample Program:

```
dir3 = 0            ' Make pin 3 an input.
REVERSE 3           ' Make pin 3 an output.
REVERSE 3           ' Make pin 3 an input.
```

SERIN

SERIN pin,baudmode,(qualifier,qualifier,...)

SERIN pin,baudmode,{#}variable,{#}variable,...

SERIN pin, baudmode, (qualifier,qualifier,...), {#}variable, {#}variable,...

Set up a serial input port and then wait for optional qualifiers and/or variables.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **Baudmode** is a variable/constant (0–7) that specifies the serial port mode. *Baudmode* can be either the # or symbol shown in the table. The other serial parameters are preset to the most common format: no parity, eight data bits, one stop bit, often abbreviated N81. These cannot be changed.

#	Symbol	Baud Rate	Polarity
0	T2400	2400	true
1	T1200	1200	true
2	T600	600	true
3	T300	300	true
4	N2400	2400	inverted
5	N1200	1200	inverted
6	N600	600	inverted
7	N300	300	inverted

- **Qualifiers** are optional variables/constants (0–255) which must be received in exact order before execution can continue.
- **Variables** (optional) are used to store received data (0–255). If any qualifiers are given, they must be satisfied before variables can be filled. If a # character precedes a variable name, then Serin will convert numeric text (e.g., numbers typed at a keyboard) into a value to fill the variable.

Serin makes the specified pin a serial input port with the characteristics set by *baudmode*. It receives serial data one byte at a time and does one of two things with it:

- Compares it to a *qualifier*.
- Stores it to a *variable*.

In either case, the Stamp will do nothing else until all qualifiers have been exactly matched in the specified order and all variables have been filled. A single Serin instruction can include both variables to fill and qualifiers to match.

SERIN *(continued)*

Here are some examples:

```
SERIN 0,T300,b2
```

Stop the program until one byte of data is received serially (true polarity, 300 baud) through pin 0. Store the received byte into variable b2 and continue. For example, if the character “A” were received, Serin would store 65 (the ASCII character code for “A”) into b2.

```
SERIN 0, T1200,#w1
```

Stop the program until a a numeric string is received serially (true polarity, 1200 baud) through pin 0. Store the value of the numeric string into variable w1. For example, suppose the following text were received: “XYZ: 576%.” Serin would ignore “XYZ: ” because these are non-numeric characters. It would collect the characters “5”, “7”, “6” up to the first non-numeric character, “%”. Serin would convert the numeric string “576” into the corresponding value 576 and store it in w1. If the # before w1 were omitted, Serin would receive only the first character, “X”, and store its ASCII character code, 88, into w1.

```
SERIN 0,N2400,("A")
```

Stop the program until a byte of data is received serially (inverted polarity, 2400 baud) through pin 0. Compare the received byte to 65, the ASCII value of the letter “A”. If it matches, continue the program. If it doesn’t match, receive another byte and repeat the comparison. The program will not continue until “A” is received. For example, if Serin received “LIMIT 65,A”, program execution would not continue until the final “A” was received.

```
SERIN 0,N2400,("SESAME"),b2,#b4
```

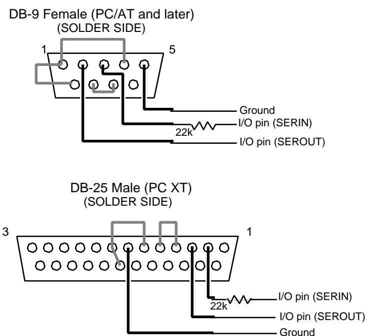
Stop the program until a string of bytes exactly matching “SESAME” is received serially (inverted polarity, 2400 baud) through pin 0. Once the qualifiers have been received, store the next byte into b2. Then receive a numeric string, convert it to a value, and store it into b4. For example, suppose Serin received, “...SESESAME! *****19*”. It would ignore the string “...SE”, then accept the matching qualifier string “SESAME”. Then Serin would put 33, the ASCII value of “!”, into b2. It would ignore the non-numeric “*” characters, then store

SERIN *(continued)*

the characters “1” and “9”. When Serin received the first non-numeric character (“*”), it would convert the text “19” into the value 19 and store it in b4. Then, having matched all qualifiers and filled all variables, Serin would permit the Stamp to go on to the next instruction.

Speed Considerations. The Serin command itself is fast enough to catch multiple bytes of data, no matter how rapidly the host computer sends them. However, if your program receives data, stores or processes it, then loops back to perform another Serin, it may miss data or receive it incorrectly because of the time delay. Use one or more of the following steps to compensate for this:

- Increase the number of stop bits at the sender from 1 to 2 (or more, if possible).
- Reduce the baud rate.
- If the sender is operating under the control of a program, add delays between transmissions.
- Reduce the amount of processing that the Stamp performs between Serins to a bare minimum.



Receiving data from a PC. To send data serially from your PC to the Stamp, all you need is a 22k resistor, some wire and connectors, and terminal communication software. Wire the connector as shown in the diagram for Serin. The wires shown in gray disable your PC’s hardware handshaking, which would normally require additional connections to control the flow of data. These aren’t required in communication with the Stamp, because you’re not likely to be sending a large volume of data as you might to a modem or printer.

When you write programs to receive serial data using this kind of

SERIN *(continued)*

hookup, make sure to specify "inverted" baudmodes, such as N2400.

If you don't have a terminal program, you can type and run the following QBASIC program to configure the serial port (2400 baud, N81) and transmit characters typed at the keyboard. QBASIC is the PC dialect of BASIC that comes with DOS versions 5 and later.

QBASIC Program to Transmit Data:

```
' This program transmits characters typed at the keyboard out the PC's
' COM1: serial port. To end the program, press control-break.
' Note: in the line below, the "0" in "CD0,CS0..." is a zero.

OPEN "COM1:9600,N,8,1,CD0,CS0,DS0,OP0" FOR OUTPUT AS #1
CLS
Again:
      theChar$ = INKEY$
      IF theChar$ = "" then Again
      PRINT #1,theChar$;
GOTO Again
```

Sample Stamp Program:

```
' To use this program, download it to the Stamp. Connect
' your PC's com1: port output to Stamp pin 0 through a 22k resistor
' as shown in the diagram. Connect a speaker to Stamp pin 7 as
' shown in the Sound entry. Run your terminal software or the QBASIC
' program above. Configure your terminal software for 2400 baud,
' N81, and turn off hardware handshaking. The QBASIC
' program is already configured this way. Try typing random
' letters and numbers--nothing will happen until you enter
' "START" exactly as it appears in the Stamp program.
' Then you may type numbers representing notes and
' durations for the Sound command. Any non-numeric text
' you type will be ignored.

      SERIN 0,N2400,("START")        ' Wait for "START".
      sound 7,(100,10)               ' Acknowledging beep.
Again:
      SERIN 0,N2400,#b2,#b3          ' Receive numeric text and
                                     ' convert to bytes.
      sound 7,(b2,b3)                ' Play corresponding sound.
      goto Again                     ' Repeat.
```

SEROUT

SEROUT pin,baudmode,({#}data,{#}data,...)

Set up a serial output port and transmit data.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **Baudmode** is a variable/constant (0–15) that specifies the serial port mode. *Baudmode* can be either the # or symbol shown in the table. The other serial parameters are preset to the most common format: no parity, eight data bits, one stop bit, often abbreviated N81. These cannot be changed.

#	Symbol	Baud Rate	Polarity and Output Mode
0	T2400	2400	true always driven
1	T1200	1200	true always driven
2	T600	600	true always driven
3	T300	300	true always driven
4	N2400	2400	inverted always driven
5	N1200	1200	inverted always driven
6	N600	600	inverted always driven
7	N300	300	inverted always driven
8	OT2400	2400	true open drain (driven high)
9	OT1200	1200	true open drain (driven high)
10	OT600	600	true open drain (driven high)
11	OT300	300	true open drain (driven high)
12	ON2400	2400	inverted open source (driven low)
13	ON1200	1200	inverted open source (driven low)
14	ON600	600	inverted open source (driven low)
15	ON300	300	inverted open source (driven low)

- **Data** are byte variables/constants (0–255) that are output by Serout. If preceded by the # sign, data items are transmitted as text strings up to five characters long. Without the #, data items are transmitted as a single byte.

Serout makes the specified pin a serial output port with the characteristics set by *baudmode*. It transmits the specified data in one of two forms:

- A single-byte value.
- A text string of one to five characters.

Here are some examples:

```
SEROUT 0,N2400,(65)
```

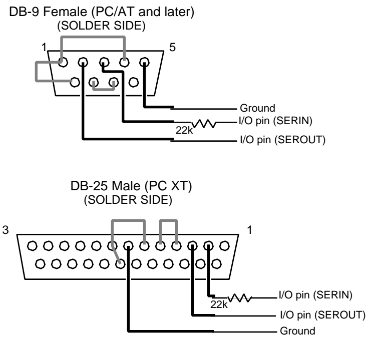
Serout transmits the byte value 65 through pin 0 at 2400 baud, inverted. If you receive this byte on a PC running terminal software, the character “A” will appear on the screen, because 65 is the ASCII code for “A”.

```
SEROUT 0,N2400,(#65)
```

SEROUT *(continued)*

Serout transmits the text string "65" through pin 0 at 2400 baud, inverted. If you receive this byte on a PC running terminal software, the text "65" will appear on the screen. When a value is preceded by the # sign, Serout automatically converts it to a form that reads correctly on a terminal screen.

When should you use the # sign? If you are sending data from the Stamp to a terminal for people to read, use #. If you are sending data to another Stamp, or to another computer for further processing, it's more efficient to omit the #.



Sending data to a PC. To send data serially to your PC from the Stamp, all you need is some wire and connectors, and terminal communication software. Wire the connector as shown in the Serout connections in the diagram at right and use the inverted baudmodes, such as N2400. Although the Stamp's serial output can only switch between 0 and +5 volts (not the ±10 volts of legal RS-232), most PCs receive it without problems.

If you don't have a terminal program, you can type and run the following QBASIC program to configure the serial port and receive characters from the Stamp.

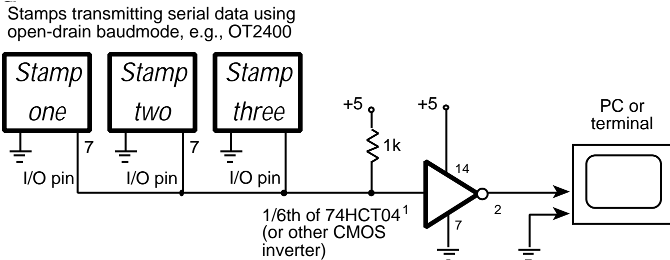
QBASIC Program to Receive Data:

```
' This program receives data transmitted by the Stamp through the PC's
' COM1: serial port and displays it on the screen. To end the program,
' press control-break. Note: in the line below, the "0" in "CD0,CS0..." is a zero.

OPEN "COM1:9600,N,8,1,CD0,CS0,DS0,OP0" FOR INPUT AS #1
CLS
Again:
	theChar$ = INPUT$(1,#1)
	PRINT theChar$;
GOTO Again
```

SEROUT *(continued)*

Open-drain/open-source signaling. The last eight configuration options for Serout begin with "O" for open-drain or open-source signaling. The diagram below shows how to use the open-drain mode to connect two or more Stamps to a common serial output line to form a network. You could also use the open-source mode, but



the resistor would have to be connected to ground, and a buffer (non-inverting driver) substituted for the inverter to drive the PC.

To understand why you must use the "open" serial modes on a network, consider what would happen if you didn't. When none of the Stamps are transmitting, all of their Serout pins are output-high. Since all are at +5 volts, no current flows between the pins. Then a Stamp transmits, and switches to output-low. With the other Stamps' pins output-high, there's a direct short from +5 volts to ground. Current flows between the pins, possibly damaging them.

If the Stamps are all set for open-drain output, it's a different story. When the Stamps aren't transmitting, their Serout pins are inputs, effectively disconnected from the serial line. The resistor to +5 volts maintains a high on the serial line. When a Stamp transmits, it pulls the serial line low. Almost no current flows through the other Stamps' Serout pins, which are set to input. Even if two Stamps transmit simultaneously, they can't damage each other.

Sample Program:

```
abc:
        pot 0,100,b2                ' Read potentiometer on pin 0.
        SEROUT 1,N300,(b2)          ' Send potentiometer
                                    ' reading over serial output.
        goto abc                    ' Repeat the process.
```

SLEEP

SLEEP seconds

Enter sleep mode for a specified number of seconds.

- **Seconds** is a variable/constant (1–65535) that specifies the duration of sleep in seconds. The length of sleep can range from 2.3 seconds (see note below) to slightly over 18 hours. Power consumption is reduced to about 20 µA, assuming no loads are being driven.

Note: The resolution of Sleep is 2.304 seconds. Sleep rounds the *seconds* up to the nearest multiple of 2.304. Sleep 1 causes 2.3 seconds of sleep, while Sleep 10 causes 11.52 seconds (5 x 2.304).

Sleep lets the Stamp turn itself off, then turn back on after a specified number of seconds. The alarm clock that wakes the Stamp up is called the watchdog timer. The watchdog is an oscillator built into the BASIC interpreter. During sleep, the Stamp periodically wakes up and adjusts a counter to determine how long it has been asleep. If it isn't time to wake up, the Stamp goes back to sleep.

To ensure accuracy of sleep intervals, the Stamp periodically compares the period of the watchdog timer to the more accurate resonator timebase. It calculates a correction factor that it uses during sleep. Longer sleep intervals are accurate to ±1 percent.

If your Stamp application is driving loads during sleep, current will be interrupted for about 18 ms when the Stamp wakes up every 2.3 seconds. The reason is that the reset that awakens the Stamp causes all of the pins to switch to input mode for approximately 18 ms. When the BASIC interpreter regains control, it restores the I/O direction dictated by your program.

If you plan to use End, Nap, or Sleep in your programs, make sure that your loads can tolerate these periodic power outages. The simplest solution is to connect resistors high or low (to +5V or ground) as appropriate to ensure a supply of current during reset.

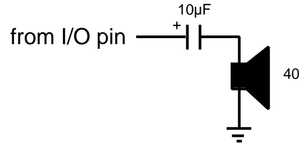
Sample Program:

```
SLEEP 3600          ' Sleep for about 1 hour.
goto xyz            ' Continue with program
                    ' after sleeping.
```

SOUND

SOUND pin,(note,duration,note,duration,...)

Change the specified pin to output, and generate square-wave notes with given durations. The output pin should be connected as shown in the diagram. You may substitute a resistor of 220 ohms or more for the capacitor, but the speaker coil will draw current even when the speaker is silent.



- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **Note(s)** are variables/constants (0–255) which specify type and frequency. Note 0 is silent for the given duration. Notes 1-127 are ascending tones. Notes 128-255 are ascending white noises, ranging from buzzing (128) to hissing (255).
- **Duration(s)** are variables/constants (1–255) which specify how long (in units of 12 ms) to play each note.

The notes produced by Sound can vary in frequency from 94.8 Hz (1) to 10,550 Hz (127). If you need to determine the frequency corresponding to a given note value, or need to find the note value that will give you best approximation for a given frequency, use the equations below.

Sample Program:

```
for b2 = 0 to 256
    SOUND 1,(25,10,b2,10)          ' Generate a constant tone (25)
                                   ' followed by an ascending tone
                                   ' (b2). Both tones have the
next                               ' same duration(10).
```

$$\textbf{Note} = 127 - \frac{\frac{1}{\textbf{Frequency (Hz)}} - 95 \times 10^{-6}}{83 \times 10^{-6}}$$

$$\textbf{Frequency (Hz)} = \frac{1}{95 \times 10^{-6} + \left((127 - \textbf{Note}) \times 83 \times 10^{-6}\right)}$$

TOGGLE

TOGGLE pin

Make pin an output and toggle state.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.

Sample Program:

```
for b2 = 1 to 25
TOGGLE 5                    'Toggle state of pin 5.
next
```

WRITE

WRITE location,data

Store data in EEPROM location.

- **Location** is a variable/constant (0–255) that specifies which EEPROM location to write to.
- **Data** is a variable/constant (0–255) that is stored in the EEPROM location.

The EEPROM is used for both program storage (which builds downward from address 254) and data storage (which builds upward from address 0). To ensure that your program doesn't overwrite itself, read location 255 in the EEPROM before writing any data. Location 255 holds the address of the first instruction in your program. Therefore, your program can use any space below the address given in location 255. For example, if location 255 holds the value 100, then your program can use locations 0–99 for data.

Sample Program:

```
read 255,b2                 ' Get location of last
                            ' program instruction.
loop:
        b2 = b2 - 1         ' Decrement to next
                            ' available EEPROM location
        serin 0,N300,b3     ' Receive serial byte in b3.
        WRITE b2,b3         ' Store received serial
                            ' byte in next EEPROM location.
if b2 > 0 then loop         ' Get another byte if there's room.
```

BLANK PAGE