Welcome to E-XFL.COM

What is "Embedded - Microcontrollers"?

"Embedded - Microcontrollers" refer to small, integrated circuits designed to perform specific tasks within larger systems. These microcontrollers are essentially compact computers on a single chip, containing a processor core, memory, and programmable input/output peripherals. They are called "embedded" because they are embedded within electronic devices to control various functions, rather than serving as standalone computers. Microcontrollers are crucial in modern electronics, providing the intelligence and control needed for a wide range of applications.

Applications of "Embedded - Microcontrollers"

Details	
Product Status	Obsolete
Core Processor	-
Core Size	-
Speed	-
Connectivity	-
Peripherals	-
Number of I/O	-
Program Memory Size	-
Program Memory Type	-
EEPROM Size	-
RAM Size	-
Voltage - Supply (Vcc/Vdd)	-
Data Converters	-
Oscillator Type	-
Operating Temperature	-
Mounting Type	-
Package / Case	-
Supplier Device Package	-
Purchase URL	https://www.e-xfl.com/product-detail/parallax/pbasic2c-ss

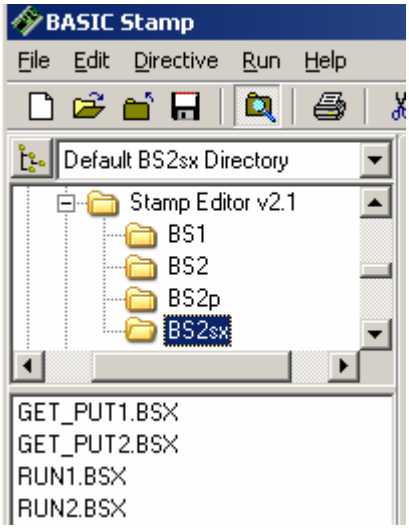


Figure 3.7: The Integrated Explorer Panel's Recent list (top), Directory list (middle), and File list (bottom).

THE DIRECTORY LIST.

The Directory list, right below the Recent list, displays drives and directories in a hierarchical tree fashion. If a directory is selected, the Folders list displays the files in that directory.

THE FILE LIST.

The File list, below the Directory list, displays all the files in the selected directory that match the selected filter (from the Filter list at the bottom. see Figure 3.8). You can select one or more files from this list and double-click, or drag-and-drop them over the editor pane, to open those files.

OPEN FROM... AND SAVE TO... OPTIONS.

You may also open files with the Open From... option by selecting File → Open From, or by pressing Ctrl+Shift+O. This allows quick access to any directory for the default and favorite directories set within Preferences (see page 60) as well as any recently used directory. The Save To... option works similarly; select File → Save To or press Ctrl+Shift+S. These features can be very helpful if you organize your files in many different directories.

THE FILTERS LIST.

The Filter list at the bottom of the explorer panel (Figure 3.8), is a drop-down list of file extension filters to apply to the File list. It works just like the "Save as type:" field of a standard Open or Save dialog.

COUNT – BASIC Stamp Command Reference

The upper-left cursor position is 0,0 (that is column 0, row 0). The right-most cursor positions depend on the size of the Debug Terminal window (which is user adjustable). If a character position that is out of range is received, the Debug Terminal wraps back around to the opposite side of the screen.

The Move To Column (x) and Move To Row (y) control characters work similarly to Move To (x,y) except they only expect a singe position value to follow.
The Clear Right (CLREOL) control character clears the characters that appear to the right of, and on, the cursor's current position. The cursor is not moved by this action.

The Clear Down (CRLDN) control character clears the characters that appear below, and on, the cursor's current line. The cursor is not moved by this action.

Name	Symbol	ASCII Value	Description
Clear Screen	CLS	0	Clear the screen and place cursor at home position.
Home	HOME	1	Place cursor at home in upper-left corner of the screen.
Move To (x,y)	CRSRXY [2.5]	2	Move cursor to specified location. Must be followed by two values (x and then y)
Cursor Left	CRSRLF[2.5]	3	Move cursor one character to left.
Cursor Right	CRSRRT[2.5]	4	Move cursor one character to right.
Cursor Up	CRSRUP[2.5]	5	Move cursor one character up.
Cursor Down	CRSRDN[2.5]	6	Move cursor one character down.
Bell	BELL	7	Beep the PC speaker.
Backspace	BKSP	8	Back up cursor to left one space.
Tab	TAB	9	Tab to the next column.
Line Feed	LF[2.5]	10	Move cursor down one line.
Clear Right	CLREOL[2.5]	11	Clear line contents to the right of cursor.
Clear Down	CLRDN[2.5]	12	Clear screen contents below cursor.
Carriage Return	CR	13	Move cursor to the first column of the next line (shift any data on the right down to that line as well).
Move To Column X	CRSRX[2.5]	14	Move cursor to specified column. Must be followed by byte value (x) for the column (0 is the left-most column).
Move To Row Y	CRSRY[2.5]	15	Move cursor to specified row. Must be followed by byte value (y) for the row (0 is the top-most row).

Table 5.13: Special DEBUG Control Characters for all BS2 models.

NOTE: ([2.5]) indicates this control character requires the PBASIC 2.5 compiler directive.

DEBUG – BASIC Stamp Command Reference

is exactly like:

```
' {$STAMP BS2}

SERIN 16, $4054, [DEC1 myNum]
```


in terms of function on a BS2. The DEBUGIN line actually takes less program space, and is obviously easier to type. Example:

```
' {$PBASIC 2.5}

myNum   VAR     Nib

DEBUG CLS, "Enter a number (1 - 5)? --> "
DEBUGIN DEC1 myNum
IF ((myNum >= 1) AND (myNum <= 5)) THEN
  DEBUG CLS, "You entered: ", DEC1 myNum
ELSE
  DEBUG CLS, "Sorry, number out of range"
ENDIF
END
```

The tables below list all the special formatters and conversion formatters available to the DEBUGIN command. See the SERIN instruction for additional information and examples of their use.

Special Formatter	Action
STR *ByteArray* \L {\E}	Input a character string of length L into an array. If specified, an end character E causes the string input to end before reaching length L. Remaining bytes are filled with 0s (zeros).
WAIT (*Value)*	Wait for a sequence of bytes specified by value. Value can be numbers separated by commas or quoted text (ex: 65, 66, 67 or "ABC"). The WAIT formatter is limited to a maximum of six characters.
WAITSTR *ByteArray* {\L}	Wait for a sequence of bytes matching a string stored in an array variable, optionally limited to L characters. If the optional L argument is left off, the end of the array-string must be marked by a byte containing a zero (0).
SKIP *Length*	Ignore *Length* bytes of characters.

Table 5.15: DEBUGIN Special Formatters.

There is an additional special formatter for the BS2p, BS2pe, and BS2px:

Special Formatter	Action
SPSTR L	Input a character string of length L bytes (up to 126) into Scratch Pad RAM, starting at location 0. Use GET to retrieve the characters.

Table 5.16: DEBUGIN Additional Special Formatter for the BS2p, BS2pe, and BS2px.

Table 5.17: DEBUGIN Conversion Formatters.

Conversion Formatter	Type of Number	Numeric Characters Accepted	Notes
DEC{1..5}	Decimal, optionally limited to 1 – 5 digits	0 through 9	1
SDEC{1..5}	Signed decimal, optionally limited to 1 – 5 digits	-, 0 through 9	1,2
HEX{1..4}	Hexadecimal, optionally limited to 1 – 4 digits	0 through 9, A through F	1,3,5
SHEX{1..4}	Signed hexadecimal, optionally limited to 1 – 4 digits	-, 0 through 9, A through F	1,2,3
IHEX{1..4}	Indicated hexadecimal, optionally limited to 1 – 4 digits	$, 0 through 9, A through F	1,3,4
ISHEX{1..4}	Signed, indicated hexadecimal, optionally limited to 1 – 4 digits	-, $, 0 through 9, A through F	1,2,3,4
BIN{1..16}	Binary, optionally limited to 1 – 16 digits	0, 1	1
SBIN{1..16}	Signed binary, optionally limited to 1 – 16 digits	-, 0, 1	1,2
IBIN{1..16}	Indicated binary, optionally limited to 1 – 16 digits	%, 0, 1	1,4
ISBIN{1..16}	Signed, indicated binary, optionally limited to 1 – 16 digits	-, %, 0, 1	1,2,4
NUM	Generic numeric input (decimal, hexadecimal or binary); hexadecimal or binary number must be indicated	$, %, 0 through 9, A through F	1, 3, 4
SNUM	Similar to NUM with value treated as signed with range -32768 to +32767	-, $, %, 0 through 9, A through F	1,2,3,4

1 All numeric conversions will continue to accept new data until receiving either the specified number of digits (ex: three digits for DEC3) or a non-numeric character.

2 To be recognized as part of a number, the minus sign (-) must immediately precede a numeric character. The minus sign character occurring in non-numeric text is ignored and any character (including a space) between a minus and a number causes the minus to be ignored.

3 The hexadecimal formatters are not case-sensitive; “a” through “f” means the same as “A” through “F”.

4 Indicated hexadecimal and binary formatters ignore all characters, even valid numerics, until they receive the appropriate prefix ($ for hexadecimal, % for binary). The indicated formatters can differentiate between text and hexadecimal (ex: ABC would be interpreted by HEX as a number but IHEX would ignore it unless expressed as $ABC). Likewise, the binary version can distinguish the decimal number 10 from the binary number %10. A prefix occurring in non-numeric text is ignored, and any character (including a space) between a prefix and a number causes the prefix to be ignored. Indicated, signed formatters require that the minus sign come before the prefix, as in -$1B45.

5 The HEX modifier can be used for Decimal to BCD Conversion. See “Hex to BCD Conversion” on page 97.

For examples of all conversion formatters and how they process incoming data, see Appendix C.

DO…LOOP

BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px

All 2

DO { WHILE | UNTIL *Condition(s)* }
** *Statement(s)***
LOOP { WHILE | UNTIL *Condition(s)* }

NOTE: DO...LOOP requires the PBASIC 2.5 compiler directive.

Function

Create a repeating loop that executes the *Statement(s)*, one or more program lines that form a code block, between DO and LOOP, optionally testing *Condition(s)* before or after the *Statement(s)*.

- ***Condition*** is an optional variable/constant/expression (0 - 65535) which determines whether the loop will run or terminate. *Condition* must follow WHILE or UNTIL.
- ***Statement*** is any valid PBASIC instruction.

Quick Facts

Table 5.18: DO...LOOP Quick Facts.

	All BS2 Models
Maximum Nested Loops	16
WHILE *Condition* Evaluation	Run loop if *Condition* evaluates as true
UNTIL *Condition* Evaluation	Terminate loop if *Condition* evaluates as true
Related Commands	FOR...NEXT and EXIT

Explanation

DO...LOOP loops let a program execute a series of instructions indefinitely or until a specified condition terminates the loop. The simplest form is shown here:

```
' {$PBASIC 2.5}

DO
  DEBUG "Error...", CR
  PAUSE 2000
LOOP
```

In this example the error message will be printed on the Debug screen every two seconds until the BASIC Stamp is reset. Simple DO...LOOP loops can be terminated with EXIT.

FOR…NEXT – BASIC Stamp Command Reference

HIGH

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px	BS2px

1 All 2

HIGH *Pin*

Function

Make the specified pin output high.

- ***Pin*** is a variable/constant/expression (0 – 15) that specifies which I/O pin to set high. This pin will be placed into output mode.

1

NOTE: Expressions are not allowed as arguments on the BS1. The range of the *Pin* argument on the BS1 is 0 – 7.

Quick Facts

Table 5.31: HIGH Quick Facts.

	BS1 and all BS2 Models
Related Commands	LOW and TOGGLE

Explanation

The HIGH command sets the specified pin to 1 (a +5 volt level) and then sets its mode to output. For example,

```
HIGH 6
```

does exactly the same thing as:

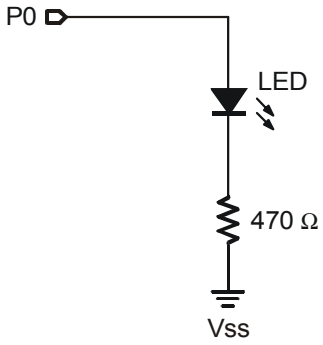
All 2

```
OUT6 = 1
DIR6 = 1
```

Using the HIGH command is faster and more concise, in this case.

Connect an LED and a resistor as shown in Figure 5.7 for demo program HIGH.bs2, below.

Figure 5.7: Example LED Circuit.



Demo Program (LOW.bs2)

```
' LOW.bs2
' This simple program sets I/O pin 0 high for 1/2 second and low for
' 1/2 second in an endless loop.  Connect an LED to P0 for a simple
' blinker.

' {$STAMP BS2}


Main:
  HIGH 0
  PAUSE 500
  LOW 0
  PAUSE 500
  GOTO Main
  END
```

NOTE: This example program can be used with the BS1 and all BS2 models by changing the $STAMP directive accordingly.

ON

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

NOTE: ON requires PBASIC 2.5.

All 2 **ON *Offset* GOTO *Address1, Address2, …AddressN***

All 2 **ON *Offset* GOSUB *Address1, Address2, …AddressN***

Function

GOTO or GOSUB to the *Address* specified by *Offset* (if in range). ON is similar in operation to BRANCH with the exception that program execution can optionally return to the line following ON (if using ON…GOSUB).

- ***Offset*** is a variable/constant/expression (0 - 255) that specifies the index (0 - N) of the address, in the list, to GOTO or GOSUB to.
- ***Address*** is a label that specifies where to go for a given *Offset*. ON will ignore any list entries beyond offset 255.

Quick Facts

Table 5.61: ON Quick Facts.

	All BS2 Models
Limit of Address Entries	256
Maximum GOSUBs per Program	255 (each ON…GOSUB counts as one GOSUB, regardless of number of address list entries)
Maximum Nested GOSUBS	4
Related Commands	BRANCH, GOTO and GOSUB

Explanation

The ON instruction is like saying, "Based ON the value of *Offset*, GOTO or GOSUB to one of these *Addresses."* ON is useful when you want to write something like this:

```
IF (value = 0) THEN GOTO Case_0        ' "GOTO" jump table
IF (value = 1) THEN GOTO Case_1
IF (value = 2) THEN GOTO Case_2
```

- or -

```
IF (value = 0) THEN GOSUB Case_0       ' "GOSUB" jump table
IF (value = 1) THEN GOSUB Case_1
IF (value = 2) THEN GOSUB Case_2
```

```
bitOne  VAR     Bit
bitTwo  VAR     Bit

bitOne = 0
bitTwo = 1
OWOUT 0, 5, [bitOne, bitTwo]
```

In the code above, we chose the value "5" for *Mode*. This sets Bit transfer and Front-End Reset modes. Also, we could have chosen to make the *bitOne* and *bitTwo* variables each a byte in size, but the BASIC Stamp would still only use the their lowest bit (BIT0) as the value to transmit in the OWOUT command (due to the *Mode* we chose).

SENDING AND FORMATTING DATA.

The OWOUT command's *OutputData* argument is similar to the DEBUG and SEROUT command's *OutputData* argument. This means data can be sent as literal text, ASCII character values, repetitive values, decimal, hexadecimal and binary translations and string data as in the examples below. (Assume a 1-wire device is used and that it transmits the string, "Value: 3A:101" every time it receives a Front-End Reset pulse).

```
value   VAR     Byte
value = 65

OWOUT 0, 1, [value]                     ' send "A"
OWOUT 0, 1, [REP value\5]               ' send "AAAAA"
OWOUT 0, 1, [DEC value]                 ' send "6" and "5"
OWOUT 0, 1, [HEX value]                 ' send "4" and "1"
OWOUT 0, 1, [BIN value]                 ' send "1000001"
```

Table 5.70 and Table 5.71 list all the special formatters and conversion formatters available to the OWOUT command. See the DEBUG and SEROUT commands for additional information and examples of their use.

Table 5.70: OWOUT Special Formatters.

Special Formatter	Action
?	Displays "symbol = x' + carriage return; where x is a number. Default format is decimal, but may be combined with conversion formatters (ex: BIN ? x to display "x = binary_number").
ASC ?	Displays "symbol = 'x'" + carriage return; where x is an ASCII character.
STR *ByteArray* {\L}	Send character string from an array. The optional \L argument can be used to limit the output to L characters, otherwise, characters will be sent up to the first byte equal to 0 or the end of RAM space is reached.
REP *Byte* \L	Send a string consisting of *Byte* repeated L times (ex: REP "X"\10 sends "XXXXXXXXXX").

Demo Program (RANDOM.bs1)

```
' RANDOM.bs1
' Connect a button to I/O pin 7 as shown in the figure in the RANDOM
' command description and run this program. This program uses RANDOM to
' simulate a coin toss. After 100 trials, it reports the total number of
' heads and tails thrown.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL  Btn             = 7                     ' button input

SYMBOL  flip            = W0                    ' a random number
SYMBOL  coin            = BIT0                  ' a bit from random number
SYMBOL  trials          = B2                    ' number of flips
SYMBOL  heads           = B3                    ' throws that come up heads
SYMBOL  tails           = B4                    ' throws that come up tails
SYMBOL  btnWrk          = B5                    ' workspace for BUTTON

Start:
  DEBUG CLS, "Press the button to toss coin.", CR

Main:
  FOR trials = 1 TO 100                         ' flip coin 100 times

Hold:
    RANDOM flip                                 ' randomize while waiting
    BUTTON Btn, 0, 250, 100, btnWrk, 0, Hold    ' wait for button press
    BRANCH coin, (Head, Tail)                   ' 0 = heads, 1 = tails

Head:
    DEBUG CR, "Heads!"
    heads = heads + 1                           ' increment heads counter
    GOTO Next_Toss

Tail:
    DEBUG CR, "Tails..."
    tails = tails + 1                           ' increment heads counter

Next_Toss:
  NEXT

  DEBUG CR, CR, "Heads: ", #heads, CR, "Tails: ", #tails
  END
```

Demo Program (RANDOM.bs2)

NOTE: This example program can be used with all BS2 models by changing the $STAMP directive accordingly.

```
' RANDOM.BS2
' Connect a button to I/O pin 7 as shown in the figure in the RANDOM
' command description and run this program. This program uses RANDOM to
' simulate a coin toss. After 100 trials, it reports the total number of
```

RCTIME

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

(See POT)

RCTIME ***Pin, State, Variable***

Function

Measure time while *Pin* remains in *State*; usually to measure the charge/discharge time of resistor/capacitor (RC) circuit.

- ***Pin*** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This pin will be placed into input mode.
- ***State*** is a variable/constant/expression (0 - 1) that specifies the desired state to measure. Once *Pin* is not in *State*, the command ends and stores the result in *Variable*.
- ***Variable*** is a variable (usually a word) in which the time measurement will be stored. The unit of time for *Variable* is described in Table 5.87.

Quick Facts

Table 5.87: RCTIME Quick Facts.

	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px
Units in *Variable*	2 µs	2 µs	0.8 µs	0.75 µs	2 µs	0.75 µs
Maximum Pulse Width	131.07 ms	131.07 ms	52.428 ms	49.151 ms	131.07 ms	49.151 ms

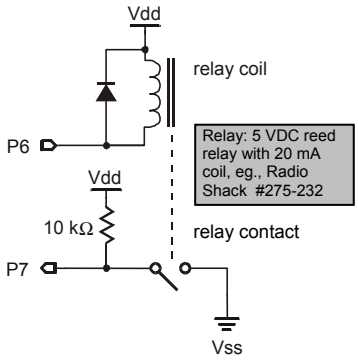
Explanation

RCTIME can be used to measure the charge or discharge time of a resistor/capacitor circuit. This allows you to measure resistance or capacitance; use R or C sensors such as thermistors or capacitive humidity sensors or respond to user input through a potentiometer. In a broader sense, RCTIME can also serve as a fast, precise stopwatch for events of very short duration.

HOW RCTIME'S TIMER WORKS.

When RCTIME executes, it makes *Pin* an input, then starts a counter (who's unit of time is shown in Table 5.87). It stops this counter as soon as the specified pin is no longer in *State* (0 or 1). If pin is not in *State* when the instruction executes, RCTIME will return 1 in *Variable*, since the instruction requires one timing cycle to discover this fact. If pin remains in *State* longer than 65535 timing cycles RCTIME returns 0.

Figure 5.34: Relay Circuit for Demo Program RCTIME2.bs2.



All 2

Demo Program (RCTIME2.bs2)

NOTE: This example program can be used with all BS2 models. This program uses conditional compilation techniques; see Chapter 3 for more information.

```
' RCTIME2.BS2
' This program illustrates the use of RCTIME as a fast stopwatch. The
' program energizes a relay coil, then measures how long it takes for the
' relay contacts to close.  The circuit for this program can be found in
' the manual. Note that RCTIME doesn't start timing instantly -- as with
' all PBASIC instructions, it must be fetched from program EEPROM before
' it can execute.

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

Coil            PIN     6
RC              PIN     7

#SELECT $STAMP
  #CASE BS2, BS2E, BS2PE
    Adjust      CON     $200            ' x 2 us per unit
  #CASE BS2SX
    Adjust      CON     $0CC            ' x 0.8 us per unit
  #CASE BS2P, BS2PX
    Adjust      CON     $0C0            ' x 0.75 us per unit
#ENDSELECT

result          VAR     Word


Main:
  DO
    LOW Coil                            ' energize relay coil
    RCTIME RC, 1, result                ' measure time to contact closure
    result = result */ Adjust           ' adjust for device
    DEBUG "Time to close: ",
```

it would see the RETURN again (although it didn't GOSUB to that routine this time) and because there wasn't a previous place to return to, the BASIC Stamp will start the entire program over again. This would cause an endless loop. The important thing to remember here is to always make sure your program doesn't allow itself to "fall into" a subroutine.

Demo Program (RETURN.bs2)

NOTE: This example program can be used with the BS1 and all BS2 models by changing the $STAMP directive accordingly.

```
' RETURN.BS2
' This program demonstrates a potential bug caused by allowing a program to
' "fall into" a subroutine.  The program was intended to indicate that it
' is "Starting...", then "Executing Subroutine,", then "Returned..." from
' the subroutine and stop.  Since we left out the END command (indicated in
' the comments), the program then falls into the subroutine, displays
' "Executing..." again and then RETURNs to the start of the program and
' runs continuously in an endless loop.

' {$STAMP BS2}


Reset:
  DEBUG "Starting Program", CR          ' show start-up

Main:
  PAUSE 1000
  GOSUB Demo_Sub                        ' call the subroutine
  PAUSE 1000
  DEBUG "Returned from Subroutine", CR  ' show that we're back
  PAUSE 1000
                                        ' <-- Forgot to put END here

Demo_Sub:
  DEBUG "  Executing Subroutine", CR    ' show subroutine activity
  RETURN
```

RUN – BASIC Stamp Command Reference

USING SERIN TO WAIT FOR SPECIFIC DATA BEFORE PROCESSING.

The SERIN command can also be configured to wait for specified data before it retrieves any additional input. For example, suppose a device that is attached to the BASIC Stamp is known to send many different sequences of data, but the only data you desire happens to appear right after the unique characters, "XYZ". The BS1 has optional *Qualifier* arguments for this purpose. On all BS2 models, a special formatter called WAIT can be used for this.

```
SYMBOL  serData = B2

SERIN 1, N2400, ("XYZ"), #serData
```

-- or --

This is written with the BS2's *Baudmode* value. Be sure to adjust the value for your BASIC Stamp.

```
serData VAR     Byte

SERIN 1, 16780, [WAIT("XYZ"), DEC serData]
```

The above code waits for the characters "X", "Y" and "Z" to be received, in that order, and then it looks for a decimal number to follow. If the device in this example were to send the characters "XYZ100" followed by a carriage return or some other non-decimal numeric character, the *serData* variable would end up with the number 100 after the SERIN line finishes. If the device sent some data other than "XYZ" followed by a number, the BASIC Stamp would continue to wait at the SERIN command.

The BS1 will accept an unlimited number of *Qualifiers*. All BS2 models will only accept up to six bytes (characters) in the WAIT formatter.

USING ASCII CODES AND CASE SENSITIVITY.

Keep in mind that when we type "XYZ" into the SERIN command, the BASIC Stamp actually uses the ASCII codes for each of those characters for its tasks. We could also have typed: 88, 89, 90 in place of "XYZ" and the code would run the same way since 88 is the ASCII code for the "X" character, 89 is the ASCII code for the "Y" character, and so on. Also note, serial communication with the BASIC Stamp is case sensitive. If the device mentioned above sent, "xYZ" or "xyZ", or some other combination of lower and upper-case characters, the BASIC Stamp would have ignored it because we told it to look for "XYZ" (all capital letters).

The BS1's SERIN command is limited to above-mentioned features. If you are not using a BS1, please continue reading about the additional features below.

STOP – BASIC Stamp Command Reference

BASIC Stamp 2px Schematic (Rev A)