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	Active
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/pbasic1-p

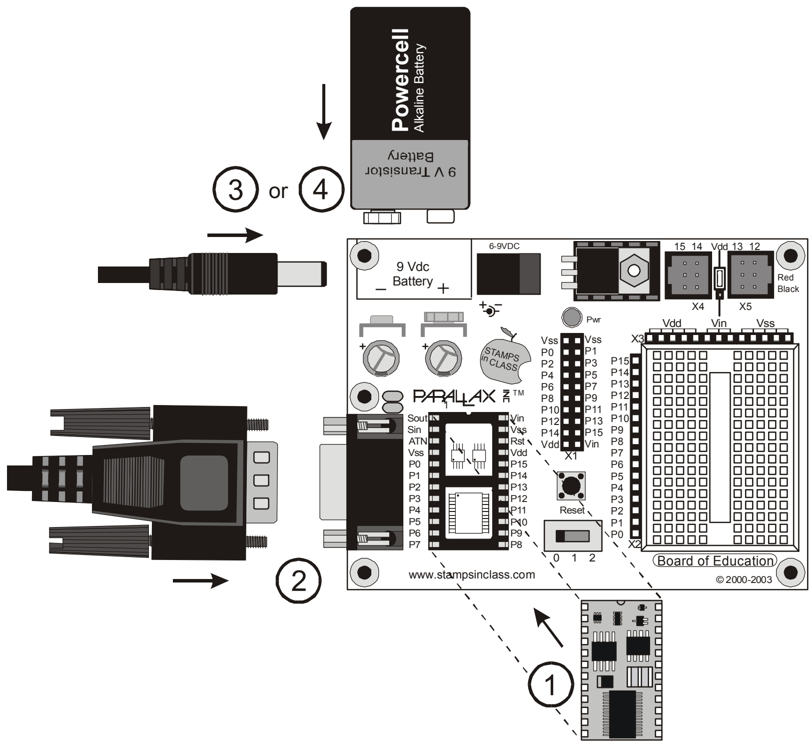


Figure 2.2: BS2-IC and Board of Education

1) Insert the BASIC Stamp module into its socket, being careful to orient it properly.

2) Connect the 9-pin female end of the serial cable to an available serial port on your computer, and then connect the male end to the Board of Education. Note: you cannot use a null modem cable.

3) Plug in the 6-9 V 300mA center-positive power supply into the barrel jack.

OR

4) Plug a 9 volt battery into the 9 VDC battery clip.

3) Install and run the BASIC Stamp Editor software.

a) If using the Parallax CD, go to the Software → BASIC Stamp → Windows section to locate the latest version. Click the Install button and follow the prompts to install and run.

b) If using the Parallax website, go to www.parallax.com → Downloads → Basic Stamp Software and look in the Software for Windows section for the latest version. Click the Download icon and follow the prompts to install and run.

c) Test your PC's connection to the BASIC Stamp by selecting Run → Identify from the menu bar, as shown in Figure 2.3. If the BASIC Stamp module is not found, check your power and cable connections and retry.

```
result          VAR     Word

result = -99                            ' Put -99 into result
                                        ' ...(2's complement format)
DEBUG SDEC ? result                     ' Display as a signed #
DEBUG SDEC ? ABS result                 ' Display as a signed #
```

COSINE: COS

All 2

The Cosine operator (COS) returns the two's complement, 16-bit cosine of an angle specified as an 8-bit "binary radian" (0 to 255) angle. COS is the same as SIN in all respects, except that the cosine function returns the x distance instead of the y distance. See "Sine: SIN", below, for a code example and more information.

DECODER: DCD

All 2

The Decoder operator (DCD) is a 2^n-power decoder of a four-bit value. DCD accepts a value from 0 to 15, and returns a 16-bit number with the bit, described by value, set to 1. For example:

```
result    VAR   Word

result  =  DCD  12                     ' Set bit 12
DEBUG  BIN16 ?  result                 ' Display result (%0001000000000000)
```

INVERSE: ~

The Inverse operator (~) complements (inverts) the bits of a number. Each bit that contains a 1 is changed to 0 and each bit containing 0 is changed to 1. This process is also known as a "bitwise NOT" and "ones complement". For example:

```
result    VAR   Byte

result = %11110001                     ' Store bits in byte result.
DEBUG  BIN8  ?  result                 ' Display in binary (%11110001)
result = ~ result                      ' Complement result
DEBUG  BIN8  ?  Result                 ' Display in binary (%00001110)
```

NEGATIVE: -

The Negative operator (-) negates a 16-bit number (converts to its twos complement).

1

```
SYMBOL  result          = W1

result = -99                           ' Put -99 into result
                                       ' ...(2's complement format)
result = result + 100                  ' Add 100 to it
DEBUG result                           ' Display result (1)
```

-- or --

Introduction

This chapter provides details on all three versions of the PBASIC Programming Language. A categorical listing of all available PBASIC commands is followed by an alphabetized command reference with syntax, functional descriptions, and example code for each command.

PBASIC LANGUAGE VERSIONS

There are three forms of the PBASIC language: PBASIC 1.0 (for the BS1), PBASIC 2.0 (for all BS2 models) and PBASIC 2.5 (for all BS2 models). You may use any version of the language that is appropriate for your BASIC Stamp module; however, when using any BS2 model, we suggest you use PBASIC 2.5 for any new programs you write because of the advanced control and flexibility it allows. PBASIC 2.5 is backward compatible with almost every existing PBASIC 2.0-based program, and code that is not 100% compatible can easily be modified to work in PBASIC 2.5.

This chapter gives details on every command for every BASIC Stamp model. Be sure to pay attention to any notes in the margins and body text regarding supported models and PBASIC language versions wherever they apply.

The BASIC Stamp Editor for Windows defaults to using PBASIC 1.0 (for the BS1) or PBASIC 2.0 (for all BS2 models). If you wish to use the default language for your BASIC Stamp model you need not do anything special. If you wish to use PBASIC 2.5, you must specify that fact, using the $PBASIC directive in your source code, for example:

```
' {$PBASIC 2.5}
```

Review the Compiler Directives section of Chapter 3 for more details on this directive. Note: you may also specify either 1.0 or 2.0 using the $PBASIC directive if you wish to explicitly state those desired languages.

Please note that the reserved word set will vary with each version of PBASIC, with additional reserved words for some BASIC Stamp models. Please see the reserved words tables in Appendix B for the complete lists. PBASIC 2.5 features many enhancements. Table 5.1 gives a brief summary of these items, with references to more information given elsewhere.

COUNT

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

All 2

COUNT ***Pin, Duration, Variable***

Function

Count the number of cycles (0-1-0 or 1-0-1) on the specified pin during the *Duration* time frame and store that number in *Variable.*

- ***Pin*** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This pin will be set to input mode.
- ***Duration*** is a variable/constant/expression (1 – 65535) specifying the time during which to count. The unit of time for *Duration* is described in Table 5.6.
- ***Variable*** is a variable (usually a word) in which the count will be stored.

Quick Facts

Table 5.6: COUNT Quick Facts.

NOTE: All timing values are approximate.

	BS2, BS2e	BS2sx	BS2p	BS2pe	BS2px
Units in *Duration*	1 ms	400 µs	287 µs	720 µs	287 µs
***Duration* range**	1 ms to 65.535 s	400 µs to 26.214 s	287 µs to 18.809 s	720 µs to 47.18 s	287 µs to 18.809 s
Minimum pulse width	4.16 µs	1.66 µs	1.20 µs	3.0 µs	1.20 µs
Maximum frequency (square wave)	120,000 Hz	300,000 Hz	416,700 Hz	166,667 Hz	416,700 Hz
Related Command	PULSIN				

Explanation

The COUNT instruction makes the *Pin* an input, then for the specified *Duration* of time, counts cycles on that pin and stores the total in *Variable*. A cycle is a change in state from 1 to 0 to 1, or from 0 to 1 to 0.

According to Table 5.6, COUNT on the BS2 can respond to transitions (pulse widths) as small as 4.16 microseconds (µs). A cycle consists of two transitions (e.g., 0 to 1, then 1 to 0), so COUNT (on the BS2) can respond to square waves with periods as short as 8.32 µs; up to 120 kilohertz (kHz) in frequency. For non-square waves (those whose high time and low time are unequal), the shorter of the high and low times must be at least 4.16 µs in

but typing the name of the variables in quotes (for the display) can get a little tedious. A special formatter, the question mark (?), can save you a lot of time. The code below does exactly the same thing (with less typing):

```
x       VAR     Byte
y       VAR     Byte

x = 100
y = 250
DEBUG DEC ? x                       ' Show decimal value of x
DEBUG DEC ? y                       ' Show decimal value of y
```

The display would look something like this:

```
x = 100
y = 250
```

The ? formatter always displays data in the form "symbol = value" (followed by a carriage return). In addition, it defaults to displaying in decimal, so we really only needed to type: DEBUG ? x for the above code. You can, of course, use any of the three number systems. For example: DEBUG HEX ? x or DEBUG BIN ? y.

It's important to note that the "symbol" it displays is taken directly from what appears to the right of the ?. If you were to use an expression, for example: DEBUG ? x*10/2+3 in the above code, the display would show: "x*10/2+3 = 503".

A special formatter, ASC, is also available for use only with the ? formatter to display ASCII characters, as in: DEBUG ASC ? x.

DISPLAYING FIXED-WIDTH NUMBERS.

What if you need to display a table of data; multiple rows and columns? The Signed/Unsigned code (above) approaches this but, if you notice, the columns don't line up. The number formatters (DEC, HEX and BIN) have some useful variations to make the display fixed-width (see Table 5.12). Up to 5 digits can be displayed for decimal numbers. To fix the value to a specific number of decimal digits, you can use DEC1, DEC2, DEC3, DEC4 or DEC5. For example:

```
x       VAR     Byte

x = 165
DEBUG DEC5 x                        ' Show decimal value of x in 5 digits
```

GOTO

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

1 All 2 **GOTO *Address***

Function

Go to the point in the program specified by *Address*.

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

Quick Facts

Table 5.30: GOTO Quick Facts.

	BS1	All BS2 Models
Related Commands	BRANCH and GOSUB	ON...GOTO, BRANCH and GOSUB
Max. GOTOs per Program	Unlimited, but good programming practices suggest using the least amount possible.	

Explanation

The GOTO command makes the BASIC Stamp execute the code that starts at the specified *Address* location. The BASIC Stamp reads PBASIC code from left to right / top to bottom, just like in the English language. The GOTO command forces the BASIC Stamp to jump to another section of code.

A common use for GOTO is to create endless loops; programs that repeat a group of instructions over and over. For example:

```
Start:
  DEBUG  "Hi", CR
GOTO  Start
```

The above code will print "Hi" on the screen, over and over again. The GOTO Start line simply tells it to go back to the code that begins with the label *Start*. Note: colons (:) are placed after labels, as in “Start:” to further indicate that they are labels, but the colon is not used on references to labels such as in the “GOTO Start” line.

1 All 2

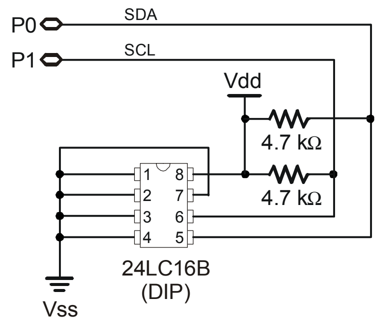
Demo Program (GOTO.bs2)

NOTE: This is written for the BS2 but can be used for the BS1 and all other BS2 models as well, by modifying the $STAMP directive accordingly.

```
' GOTO.bs2
' This program is not very practical, but demonstrates the use of GOTO to
' jump around the code.  This code jumps between three different routines,
' each of which print something different on the screen.  The routines are
' out of order for this example.
```

Figure 5.8: Example Circuit for the I2CIN command and a 24LC16B EEPROM.

Note: The 4.7 kΩ resistors are required for the I2CIN command to function properly.



RECEIVING FORMATTED DATA.

The I2CIN command's *InputData* argument is similar to the SERIN command's *InputData* argument. This means data can be received as ASCII character values, decimal, hexadecimal and binary translations and string data as in the examples below. (Assume the 24LC16B EEPROM is used and it has the string, "Value: 3A:101" stored, starting at location 0).

```
value   VAR     Byte(13)

I2CIN 0, $A1, 0, [value]            ' receive the ASCII value for "V"
I2CIN 0, $A1, 0, [DEC value]        ' receive the number 3
I2CIN 0, $A1, 0, [HEX value]        ' receive the number $3A
I2CIN 0, $A1, 0, [BIN value]        ' receive the number %101
I2CIN 0, $A1, 0, [STR value\13]     ' receive the string "Value: 3A:101"
```

Table 5.33 and Table 5.34 below list all the available special formatters and conversion formatters available to the I2CIN command. See the SERIN command for additional information and examples of their use.

Table 5.33: I2CIN Special Formatters.

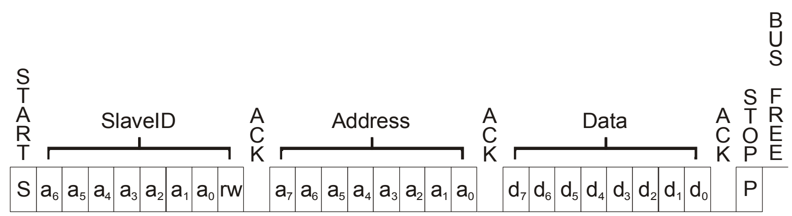
Special Formatter	Action
SKIP *Length*	Ignore *Length* bytes of characters.
SPSTR L	Input a character stream of length L bytes (up to 126) into Scratch Pad RAM, starting at location 0. Use GET to retrieve the characters.
STR *ByteArray* \L {\E}	Input a character string of length L into an array. If specified, an end character E causes the string input to end before reaching length L. Remaining bytes are filled with 0s (zeros).
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).

depending on the I^{2}C device. Note that every device has different limitations regarding how may contiguous bytes they can receive or transmit in one session. Be aware of this, and program accordingly.

START AND STOP CONDITIONS AND ACKNOWLEDGMENTS.

Every I^{2}C transmission session begins with a Start Condition and ends with a Stop Condition. Additionally, immediately after every byte is transmitted, an extra clock cycle is used to send or receive an acknowledgment signal (ACK). All of these operations are automatically taken care of by the I2COUT command so that you need not be concerned with them. The general I^{2}C transmission format is shown in Figure 5.13.

Figure 5.13: I^{2}C Transmission Format



NOTES:
S = Start Condition
P = Stop Condition
a = id or address bit
d = data bit (transmitted by the BASIC Stamp or the I^{2}C device)
ACK = Acknowledge signal. (Most acknowledge signals are generated by the I^{2}C device)

SPECIAL NOTE ABOUT I2COUT INPLIMENTATION.

Since the I2COUT command is intended for output only, it actually overrides the "R/W" bit (bit 0) in the *SlaveID* argument. This is done to avoid device conflicts should the value be mistyped. Put simply, this means commands such as: I2COUT 0, $A0, 10, [0] and I2COUT 0, $A1, 10, [0] both transmit the same thing ($A0, then 10, then the data). Even though the I2COUT command really doesn't care what the value of the *SlaveID*'s LSB is, it is suggested that you still set it appropriately for clarity.

Also note that the I2COUT command does not support multiple I^{2}C masters and the BASIC Stamp cannot operate as an I^{2}C slave device.

Demo Program (I2C.bsp)

NOTE: This example program can be used with the BS2p, BS2pe and BS2px. This program uses conditional compilation techniques; see Chapter 3 .

```
' I2C.bsp
' This program demonstrates writing and reading every location in a 24LC16B
' EEPROM using the BS2p/BS2pe's I2C commands.  Connect the BS2p, BS2pe, or
' BS2px to the 24LC16B DIP EEPROM as shown in the diagram in the I2CIN or
```

LOOKDOWN

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

1 **LOOKDOWN *Target*, (*Value0, Value1, ...ValueN*), *Variable***

All 2 **LOOKDOWN *Target*, { *ComparisonOp* } [*Value0, Value1, ...ValueN*], *Variable***

Function

Compare *Target* value to a list of values and store the index number of the first value that matches into *Variable*. If no value in the list matches, *Variable* is left unaffected. On all BS2 models, the optional *ComparisonOp* is used as criteria for the match; the default criteria is "equal to."

1 NOTE: Expressions are not allowed as arguments on the BS1.

- ***Target*** is a variable/constant/expression (0 – 65535) to be compared to the values in the list.

All 2

- ***ComparisonOp*** is an optional comparison operator (as described in Table 5.53) to be used as the criteria when comparing values. When no *ComparisonOp* is specified, equal to (=) is assumed. This argument is not available on the BS1.
- ***Values*** are variables/constants/expressions (0 – 65535) to be compared to *Target*.
- ***Variable*** is a variable (usually a byte) that will be set to the index (0 – 255) of the matching value in the *Values* list. If no matching value is found, *Variable* is left unaffected.

Quick Facts

Table 5.52: LOOKDOWN Quick Facts.

	BS1 and all BS2 Models
Limit of *Value* Entries	256
Starting Index Number	0
If value list contains no match…	Variable is left unaffected
Related Command	LOOKUP

Explanation

LOOKDOWN works like the index in a book. In an index, you search for a topic and get the page number. LOOKDOWN searches for a target value in a list, and stores the index number of the first match in a variable. For example:

All 2

```
value      VAR   Byte
result     VAR   Nib

value   =  "f"
result  =  255

LOOKDOWN  value, ["The quick brown fox"], result
DEBUG  "Value matches item ", DEC  result, " in list"
```

DEBUG prints, "Value matches item 16 in list" because the character at index item 16 is "f" in the phrase, "The quick brown fox".

LOOKDOWN CAN USE VARIABLES AND EXPRESSIONS IN THE VALUE LIST.

The examples above show LOOKDOWN working with lists of constants, but it also works with variables and expressions also. Note, however, that expressions are not allowed as argument on the BS1.

USING LOOKDOWN'S COMPARISON OPERATORS.

All 2

On all BS2 models, the LOOKDOWN command can also use another criteria (other than "equal to") for its list. All of the examples above use LOOKDOWN's default comparison operator, =, that searches for an exact match. The entire list of *ComaprisonOps* is shown in Table 5.53. The "greater than" comparison operator (>) is used in the following example:

```
value      VAR   Byte
result     VAR   Nib

value   =  17
result  =  15

LOOKDOWN  value, >[26, 177, 13, 1, 0, 17, 99], result
DEBUG  "Value greater than item ", DEC  result, " in list"
```

DEBUG prints, "Value greater than item 2 in list" because the first item the value 17 is greater than is 13 (which is item 2 in the list). *Value* is also greater than items 3 and 4, but these are ignored, because LOOKDOWN only cares about the first item that matches the criteria. This can require a certain amount of planning in devising the order of the list. See the demo program below.

WATCH OUT FOR UNSIGNED MATH ERRORS WHEN USING THE COMPARISON OPERATORS.

LOOKDOWN comparison operators (Table 5.53) use unsigned 16-bit math. They will not work correctly with signed numbers, which are represented internally as two's complement (large 16-bit integers). For example, the two's complement representation of -99 is 65437. So although -99 is certainly less than 0, it would appear to be larger than zero

Demo Program (POLLRUN0.bsp)

NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the $STAMP directive accordingly.

```
' POLLRUN0.bsp
' This program demonstrates the POLLRUN command.  It is intended to be
' downloaded to program slot 0, and the program called POLLRUN1.bsp
' should be downloaded to program slot 1.  I/O pin 0 is set to watch for
' a low signal.  Once the Main routine starts running, the program
' continuously prints it's program slot number to the screen.  If I/O
' pin 0 goes low, the program in program slot 1 (which should be
' POLLRUN1.bsp) is run.

' {$STAMP BS2p, POLLRUN1.BSP}
' {$PBASIC 2.5}

pgmSlot         VAR     Byte

Setup:
  POLLIN  0, 0                          ' polled-input, look for 0
  POLLRUN 1                             ' run slot 1 on polled activation
  POLLMODE 3                            ' enable polling

Main:
  GET 127, pgmSlot
  DEBUG "Running Program #", DEC pgmSlot.LOWNIB, CR
  GOTO Main
  END
```

Demo Program (POLLRUN1.bsp)

NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the $STAMP directive accordingly.

```
' POLLRUN1.bsp
' This program demonstrates the POLLRUN command.  It is intended to be
' downloaded to program slot 1, and the program called POLLRUN0.bsp
' should be downloaded to program slot 0.  This program is run when
' program 0 detects a low on I/O pin 0 via the polled commands.

' {$STAMP BS2p}
' {$PBASIC 2.5}

pgmSlot         VAR     Byte

Main:
  GET 127, pgmSlot
  DEBUG "Running Program #", DEC pgmSlot.LOWNIB, CR
  GOTO Main
  END
```

1

```
SYMBOL  result          = W1

Setup:
  result = 11000

Main:
  RANDOM result
  DEBUG result
  GOTO Main
```

-- or --

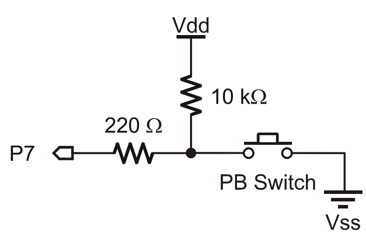
All 2

```
result          VAR     Word

Setup:
  result = 11000

Main:
  RANDOM result
  DEBUG DEC ? result
  GOTO Main
```

Here, *result* is only initialized once, before the loop. Each time through the loop, the previous value of *result*, generated by RANDOM, is used as the next seed value. This generates a more desirable set of pseudorandom numbers.

In applications requiring more apparent randomness, it's necessary to "seed" RANDOM with a more random value every time. For instance, in the demo program below, RANDOM is executed continuously (using the previous resulting number as the next seed value) while the program waits for the user to press a button. Since the user can’t control the timing of button presses very accurately, the results approach true randomness.

Figure 5.32: RANDOM Button Circuit.



All 2

Demo Program (READ.bs2)

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

```
' READ.bs2
' This program reads a string of data stored in EEPROM. The EEPROM data is
' downloaded to the BS2 at compile-time and remains there (even with the
' power off) until overwritten. Put ASCII characters into EEPROM, followed
' by 0, which will serve as the end-of-message marker.

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

strAddr         VAR     Word
char            VAR     Byte

Msg1            DATA    "BS2", CR, "EEPROM Storage!", 0


Main:
  strAddr = Msg1                        ' set to start of message
  GOSUB String_Out
  END

String_Out:
  DO
    READ strAddr, char                  ' read byte from EEPROM
    strAddr = strAddr + 1               ' point to next character
    IF (char = 0) THEN EXIT             ' if 0, exit routine
    DEBUG char                          ' otherwise print char
  LOOP
  RETURN
```

```
  FOR idx = 0 TO 4                            ' read/display table values
    READ (slotNum * 5) + idx, value
    DEBUG DEC3 value, " "
  NEXT
  DEBUG CR
  PAUSE 1000

  RUN 1                                       ' run Slot 1 pgm
```

Demo Program (RUN2.bsx)

NOTE: This example program was written for the BS2sx but can be used with the BS2e, BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

```
' RUN2.bsx
' This example demonstrates the use of the RUN command.  First, the SPRAM
' location that holds the current slot is read using the GET command to
' display the currently running program number.  Then a set of values
' (based on the program number) are displayed on the screen.  Afterwards,
' program number 0 is run. This program is a BS2sx project consisting of
' RUN1.BSX and RUN2.BSX, but will run on all multi-slot BASIC Stamp models.

' {$STAMP BS2sx}
' {$PBASIC 2.5}

#SELECT $STAMP                                ' set SPRAM of slot number
  #CASE BS2
    #ERROR "Multi-slot BASIC Stamp required."
  #CASE BS2E, BS2SX
    Slot        CON     63
  #CASE BS2P, BS2PE, BS2PX
    Slot        CON     127
#ENDSELECT

slotNum         VAR     Nib                   ' current slot
idx             VAR     Nib                   ' loop counter
value           VAR     Byte                  ' value from EEPROM

EEtable         DATA    100, 40, 80, 32, 90
                DATA    200, 65, 23, 77, 91


Setup:
  GET Slot, slotNum                           ' read current slot
  DEBUG  "Program #", DEC slotNum, CR         ' display

Main:
  FOR idx = 0 TO 4                            ' read/display table values
    READ (slotNum * 5) + idx, value
    DEBUG DEC3 value, " "
  NEXT
  DEBUG CR
  PAUSE 1000

  RUN 0                                       ' back to Slot 0 pgm
```

Table 5.96: *Baudmode* calculation for all BS2 models. Add the results of steps 1, 2 and 3 to determine the proper value for the *Baudmode* argument.

Step 1: Determine the bit period (bits 0 – 11).	BS2, BS2e and BS2pe: = INT(1,000,000 / baud rate) – 20 BS2sx and BS2p: = INT(2,500,000 / baud rate) – 20 BS2px: = INT(4,000,000 / baud rate) – 20 Note: INT means 'convert to integer;' drop the numbers to the right of the decimal point.
Step 2: Set data bits and parity (bit 13).	8-bit/no-parity = 0 7-bit/even-parity = 8192
Step 3: Select polarity (bit 14).	True (noninverted) = 0 Inverted = 16384

Table 5.97: BS2, BS2e, and BS2pe common baud rates and corresponding *Baudmodes*.

Baud Rate	8-bit no-parity inverted	8-bit no-parity true	7-bit even-parity inverted	7-bit even-parity true
300	19697	3313	27889	11505
600	18030	1646	26222	9838
1200	17197	813	25389	9005
2400	16780	396	24972	8588
4800*	16572	188	24764	8380
9600*	16468	84	24660	8276

*The BS2, BS2e and BS2pe may have trouble synchronizing with the incoming serial stream at this rate and higher due to the lack of a hardware input buffer. Use only simple variables and no formatters to try to solve this problem.

Table 5.98: BS2sx and BS2p common baud rates and corresponding *Baudmodes.*

Baud Rate	8-bit no-parity inverted	8-bit no-parity true	7-bit even-parity inverted	7-bit even-parity true
1200	18447	2063	26639	10255
2400	17405	1021	25597	9213
4800*	16884	500	25076	8692
9600*	16624	240	24816	8432

*The BS2sx and BS2p may have trouble synchronizing with the incoming serial stream at this rate and higher due to the lack of a hardware input buffer. Use only simple variables and no formatters to try to solve this problem.

Table 5.99: BS2px common baud rates and corresponding *Baudmodes.*

Baud Rate	8-bit no-parity inverted	8-bit no-parity true	7-bit even-parity inverted	7-bit even-parity true
1200	19697	3313	27889	11505
2400	18030	1646	26222	9838
4800	17197	813	25389	9005
9600	16780	396	24792	8588

CHOOSING THE PROPER BAUD MODE.

If you're communicating with existing software or hardware, its speed(s) and mode(s) will determine your choice of baud rate and mode. In general, 7-bit/even-parity (7E) mode is used for text, and 8-bit/no-parity (8N) for byte-oriented data. Note: the most common mode is 8-bit/no-parity, even when the data transmitted is just text. Most devices

```
    T38K4       CON     6
  #CASE BS2SX, BS2P
    T1200       CON     2063
    T2400       CON     1021
    T9600       CON     240
    T19K2       CON     110
    T38K4       CON     45
  #CASE BS2PX
    T1200       CON     3313
    T2400       CON     1646
    T9600       CON     396
    T19K2       CON     188
    T38K4       CON     84
#ENDSELECT

Inverted        CON     $4000
Open            CON     $8000
Baud            CON     T38K4 + Inverted


Main:
  DO
    SEROUT SO\FC, Baud, ["Hello!", CR]  ' send the greeting
    PAUSE 2500                          ' wait 2.5 seconds
  LOOP                                  ' repeat forever
  END
```

All 2

Demo Program (SERIN_SEROUT2.bs2)

NOTE: This example program was written for the BS2 but it can be used with the BS2e, BS2sx, BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

```
' SERIN_SEROUT2.bs2
' Using two BS2-IC's, connect the circuit shown in the SERIN command
' description and run this program on the BASIC Stamp designated as the
' Receiver.  This program demonstrates the use of Flow Control (FPin).
' Without flow control, the sender would transmit the whole word "Hello!"
' in about 1.5 ms. The receiver would catch the first byte at most; by the
' time it got back from the first 1-second PAUSE, the rest of the data
' would be long gone. With flow control, communication is flawless since
' the sender waits for the receiver to catch up.

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

SI              PIN     0               ' serial input
FC              PIN     1               ' flow control pin

#SELECT $STAMP
  #CASE BS2, BS2E, BS2PE
    T1200       CON     813
    T2400       CON     396
    T9600       CON     84
    T19K2       CON     32
    T38K4       CON     6
```

Demo Program (STORE1.bsp)

NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the $STAMP directive accordingly.

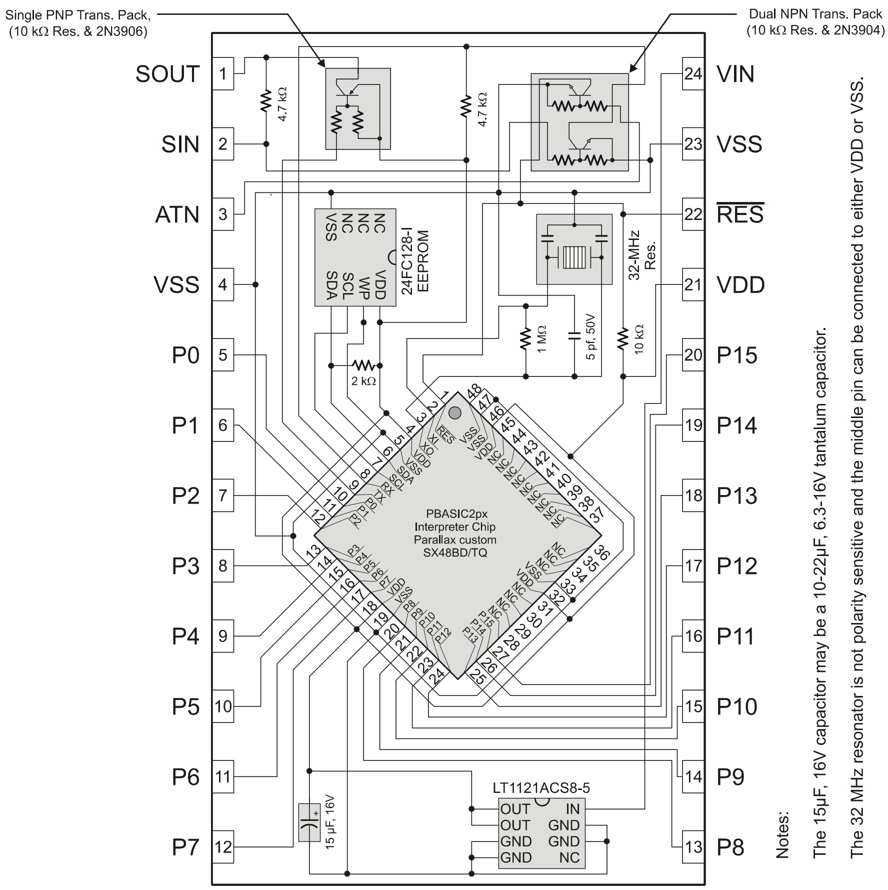
```
' STORE1.bsp

' {$STAMP BS2p}
' {$PBASIC 2.5}

idx             VAR     Word            ' index
value           VAR     Byte

LocalData       DATA    @0, 6, 7, 8, 9, 10


Main:
  GOSUB Show_Slot_Info                  ' show slot info/data
  PAUSE 2000
  STORE 0                               ' point READ/WRITE to Slot 0
  GOSUB Show_Slot_Info
  PAUSE 2000
  RUN 2                                 ' run program in Slot 2
  END

Show_Slot_Info:
  GET 127, value
  DEBUG CR, "Pgm Slot: ", DEC value.NIB0,
        CR, "R/W Slot: ", DEC value.NIB1,
        CR, CR

  FOR idx = 0 TO 4
    READ idx, value
    DEBUG "Location: ", DEC idx, TAB,
          "Value: ", DEC3 value, CR
  NEXT
  RETURN
```

Demo Program (STORE2.bsp)

NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the $STAMP directive accordingly.

```
' STORE2.bsp

' {$STAMP BS2p}
' {$PBASIC 2.5}

idx             VAR     Word            ' index
value           VAR     Byte

LocalData       DATA    @0, 11, 12, 13, 14, 15


Main:
  GOSUB Show_Slot_Info                  ' show slot info/data
  PAUSE 2000
  STORE 0                               ' point READ/WRITE to Slot 0
```

BASIC Stamp 2px Schematic (Rev A)



Notes:

The 15µF, 16V capacitor may be a 10-22µF, 6.3-16V tantalum capacitor.

The 32 MHz resonator is not polarity sensitive and the middle pin can be connected to either VDD or VSS.