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/pbasic2sx-28-ss

Internet BASIC Stamp Discussion List

We maintain active web-based discussion forums for people interested in Parallax products. These lists are accessible from www.parallax.com via the Support → Discussion Forums menu. These are the forums that we operate from our web site:

- BASIC Stamps – This list is widely utilized by engineers, hobbyists and students who share their BASIC Stamp projects and ask questions.
- Stamps in Class® – Created for educators and students, subscribers discuss the use of the Stamps in Class curriculum in their courses. The list provides an opportunity for both students and educators to ask questions and get answers.
- Parallax Educators –Exclusively for educators and those who contribute to the development of Stamps in Class. Parallax created this group to obtain feedback on our curricula and to provide a forum for educators to develop and obtain Teacher's Guides.
- Translators – The purpose of this list is to provide a conduit between Parallax and those who translate our documentation to languages other than English. Parallax provides editable Word documents to our translating partners and attempts to time the translations to coordinate with our publications.
- Robotics – Designed exclusively for Parallax robots, this forum is intended to be an open dialogue for a robotics enthusiasts. Topics include assembly, source code, expansion, and manual updates. The Boe-Bot®, Toddler®, SumoBot®, HexCrawler and QuadCrawler robots are discussed here.
- SX Microcontrollers and SX-Key – Discussion of programming the SX microcontroller with Parallax assembly language SX – Key® tools and 3rd party BASIC and C compilers.
- Javelin Stamp – Discussion of application and design using the Javelin Stamp, a Parallax module that is programmed using a subset of Sun Microsystems' Java® programming language.

Supported Hardware, Firmware and Software

This manual is valid with the following software and firmware versions:

BASIC Stamp Model	Firmware	Windows Interface
BASIC Stamp 1	1.4	2.2
BASIC Stamp 2	1.0	2.2
BASIC Stamp 2e	1.1	2.2
BASIC Stamp 2sx	1.1	2.2
BASIC Stamp 2p	1.4	2.2
BASIC Stamp 2pe	1.1	2.2
BASIC Stamp 2px	1.0	2.2

The information herein will usually apply to newer versions but may not apply to older versions. New software can be obtained free on web site (www.parallax.com). If you have any questions about what you need to upgrade your product, please contact Parallax.

Credits

Authorship and Editorial Review Team: Jeff Martin, Jon Williams, Ken Gracey, Aristides Alvarez, and Stephanie Lindsay; Cover Art: Jen Jacobs; Technical Graphics, Rich Allred; with many thanks to everyone at Parallax Inc.

#ERROR SYNTAX.

#ERROR *Message*

#ERROR displays a compile-time error. This allows the programmer to flag fatal errors during compilation.

- ***Message*** is the error message string, enclosed in quotes.

Example:

```
' {$PBASIC 2.5}

#DEFINE I2CReady = (($STAMP = BS2p) OR ($STAMP = BS2pe) OR ($STAMP = BS2px))

#IF NOT I2CReady #THEN
  #ERROR "BS2p, BS2pe, or BS2px is required for this program."
#ENDIF
```

When compiled, this example will cause the editor to halt compilation and display the dialog below if you attempt to compile for a BASIC Stamp model other than the BS2p, BS2pe, or BS2px:

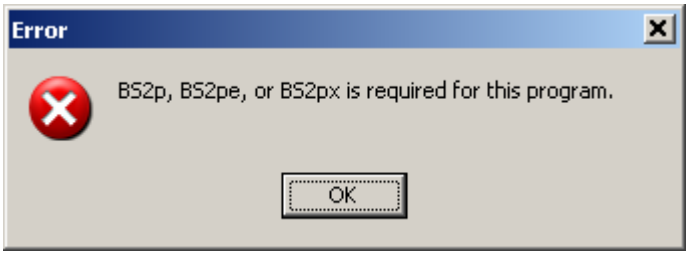


Figure 3.27: Custom Error Message using the #ERROR directive.

Features for Developers

The BASIC Stamp Editor has several features that are designed to support the needs of developers. Note: when installing the BASIC Stamp editor, you can instruct the installer to include additional developer resources by selecting the "Custom" option from the "Setup Type" prompt.

GENERATE OBJECT CODE FEATURE.

The Generate Object Code feature allows you to tokenize a PBASIC program and save it to a file in the tokenized form. This allows you to send your BASIC Stamp object code (the actual binary data that is downloaded to the BASIC Stamp module) to other people without having to reveal your PBASIC source code. If you are a developer who has

Using the BASIC Stamp Editor

The INS variable always shows the state of the I/O pins themselves, regardless of the direction of each I/O pin. We call this, "reading the pins". If a pin was set to an input mode (within DIRS) and an external circuit connected the I/O pin to ground, the corresponding bit of INS would be low. If a pin was set to an output mode and the pin's state was set to a high level (within OUTS), the corresponding bit of INS would be high. If, however, that same pin was externally connected directly to ground, the corresponding bit of INS would be low; since we're reading the state of the pin itself and the BASIC Stamp cannot override a pin that is driven to ground or 5 volts externally. Note: The last example is an error, is a direct short and can cause damage to the BASIC Stamp! Do not intentionally connect output pins directly to an external power source or you risk destroying your BASIC Stamp.

SUMMARY OF THE FUNCTION OF DIRS, INS AND OUTS.

To summarize: DIRS determines whether a pin's state is set by external circuitry (input, 0) or by the state of OUTS (output, 1). INS always matches the actual states of the I/O pins, whether they are inputs or outputs. OUTS holds bits that will only appear on pins whose DIRS bits are set to output.

In programming the BASIC Stamp, it's often more convenient to deal with individual bytes, nibbles or bits of INS, OUTS and DIRS rather than the entire 16-bit words. PBASIC has built-in names for these elements, shown in Table 4.2.

Here's an example of what is described in Table 4.2. The INS register is 16-bits (corresponding to I/O pins 0 though 15). The INS register consists of two bytes, called INL (the Low byte) and INH (the High byte). INL corresponds to I/O pins 0 through 7 and INH corresponds to I/O pins 8 though 15. INS can also be thought of as containing four nibbles, INA, INB, INC and IND. INA is I/O pins 0 though 3, INB is I/O pins 4 though 7, etc. In addition, each of the bits of INS can be accessed directly using the names IN0, IN1, IN2… IN5.

The same naming scheme holds true for the OUTS and DIRS variables as well.

PREDEFINED "FIXED" VARIABLES.

As Table 4.2 shows, the BASIC Stamp module's memory is organized into 16 words of 16 bits each. The first three words are used for I/O. The remaining 13 words are available for use as general-purpose variables.

ORDER OF OPERATIONS.

The BASIC Stamp solves math problems in the order they are written: from left to right. The result of each operation is fed into the next operation. So to compute 12+3*2/4, the BASIC Stamp goes through a sequence like this:

```
12 + 3 = 15
15 * 2 = 30
30 / 4 = 7
```

Since the BASIC Stamp performs integer math (whole numbers only) 30 / 4 results in 7, not 7.5. We'll talk more about integers in the next section.

Some other dialects of BASIC would compute that same expression based on their precedence of operators, which requires that multiplication and division be done before addition. So the result would be:

```
3 * 2 = 6
6 / 4 = 1
12 + 1 = 13
```

Once again, because of integer math, the fractional portion of 6 / 4 is dropped, so we get 1 instead of 1.5.

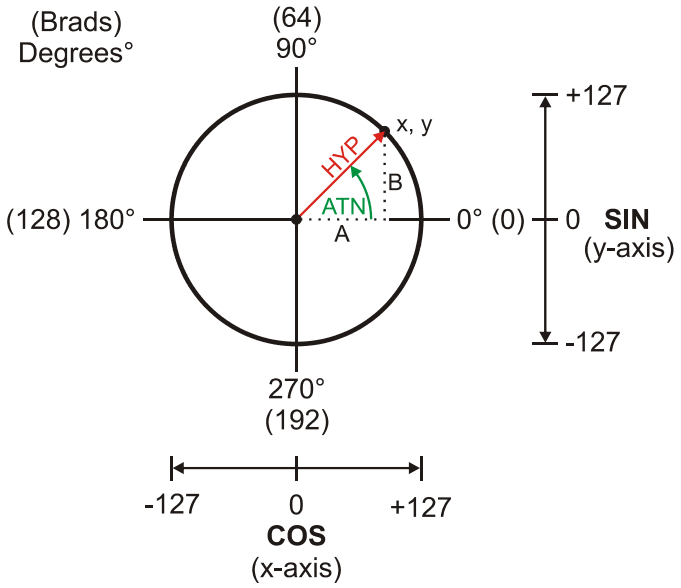
The BS1 does not allow parenthesis in expressions. Unfortunately, all expressions have to be written so that they evaluate as intended strictly from left to right.

All BS2 models, however, allow parentheses to be used to change the order of evaluation. Enclosing a math operation in parentheses gives it priority over other operations. To make the BASIC Stamp compute the previous expression in the conventional way, you would write it as 12 + (3*2/4). Within the parentheses, the BASIC Stamp works from left to right. If you wanted to be even more specific, you could write 12 + ((3*2)/4). When there are parentheses within parentheses, the BASIC Stamp works from the innermost parentheses outward. Parentheses placed within parentheses are called "nested parentheses."

INTEGER MATH.

The BASIC Stamp performs all math operations by the rules of positive integer math. That is, it handles only whole numbers, and drops any fractional portions from the results of computations. The BASIC Stamp handles negative numbers using two's complement rules.

Figure 4.2: ATN and HYP operators in the PBASIC unit circle



HYPOTENUSE: HYP

All 2

The Hypotenuse operator (HYP) returns the length of the hypotenuse of a right triangle with sides of length A and B. The syntax of HYP is:

```
SideA HYP SideB
```

where SideA and SideB are the side lengths of a right-triangle (the order isn't important). Another application of HYP is to calculate the distance between the origin (0, 0) and a point (X, Y) in a Cartesian coordinate system. Side length (vector) input values are limited to -127 to 127 (signed bytes). See diagram with ATN operator, Figure 4.2.

```
DEBUG ? 3 HYP 4                 ' hypotenuse of 3x4 triangle (5)
```

MINIMUM: MIN

1 All 2

The Minimum operator (MIN) limits a value to a specified 16-bit positive minimum. The syntax of MIN is:

```
value  MIN  limit
```

where *value* is a constant or variable value to perform the MIN function upon and *limit* is the minimum value that *value* is allowed to be. Its logic is, 'if *value* is less than *limit*, then make result = *limit*; if *value* is greater than or equal to *limit*, make result = *value*.'

DATA

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

(See EEPROM)

{ *Symbol* } **DATA** *DataItem* { , *DataItem...* }

Function

Write data to the EEPROM during program download.

- ***Symbol*** is an optional, unique symbol name that will be automatically defined as a constant equal to the location number of the first data item.
- ***DataItem*** is a constant/expression (0 – 65535) indicating a value, and optionally how to store the value.

Quick Facts

Table 5.7: DATA Quick Facts.

	All BS2 Models
Special Notes	Writes values to EEPROM during download in blocks of 16 bytes. Writes byte or word-sized values. Can be used to decrease program size.
Related Commands	READ and WRITE

Explanation

When you download a program into the BASIC Stamp, it is stored in the EEPROM starting at the highest address (2047) and working towards the lowest address. Most programs don't use the entire EEPROM, so the lower portion is available for other uses. The DATA directive allows you to define a set of data to store in the available EEPROM locations. It is called a "directive" rather than a "command" because it performs an activity at compile-time rather than at run-time (i.e.: the DATA directive is not downloaded to the BASIC Stamp, but the data it contains is downloaded).

WRITING SIMPLE, SEQUENTIAL DATA.

The simplest form of the DATA directive is something like the following:

```
DATA        100, 200, 52, 45
```

This example, when downloaded, will cause the values 100, 200, 52 and 45 to be written to EEPROM locations 0, 1, 2 and 3, respectively. You can then use the READ and WRITE commands in your code to access these locations and the data you've stored there.

INTERNAL REPRESENTATION OF BOOLEAN VALUES (TRUE VS. FALSE).

Internally, the BASIC Stamp defines "false" as 0 and "true" as any value other than 0. Consider the following instructions:

All 2

```
flag    VAR     Bit

Setup:
  flag = 1

Test:
  IF flag THEN Is_True
  DEBUG  "False"
  END

Is_True:
  DEBUG "True"
  END
```

Since *flag* is 1, IF...THEN would evaluate it as true and print the message "True" on the screen. Suppose you changed the IF...THEN command to read "IF NOT flag THEN Is_True." That would also evaluate as true. Whoa! Isn't NOT 1 the same thing as 0? No, at least not in the 16-bit world of the BASIC Stamp.

Internally, the BASIC Stamp sees a bit variable containing 1 as the 16-bit number %0000000000000001. So it sees the NOT of that as %1111111111111110. Since any non-zero number is regarded as true, NOT 1 is true. Strange but true.

AVOIDING ERRORS WITH BOOLEAN RESULTS.

The easiest way to avoid the kinds of problems this might cause is to always use a conditional operator with IF...THEN. Change the example above to read IF flag = 1 THEN is_True. The result of the comparison will follow IF...THEN rules. Also, the logical operators will work as they should; IF NOT Flag = 1 THEN is_True will correctly evaluate to false when *flag* contains 1.

This also means that you should only use the "named" conditional logic operators NOT, AND, OR, and XOR with IF...THEN. The conditional logic operators format their results correctly for IF...THEN instructions. The other logical operators, represented by symbols ~, &, |, and ^ do not; they are binary logic operators.

All 2

The remainder of this discussion only applies to the extended IF…THEN syntax supported by PBASIC 2.5.

LCDOUT

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

LCDOUT ***Pin*****,** ***Command*****, [** ***OutputData*** **]**

Function

Send data to an LCD display.

- ***Pin*** is a variable/constant/expression (0 – 1 or 8 – 9) that specifies which I/O pins to use. The LCD requires, at most, seven I/O pins to operate. The *Pin* argument serves a double purpose; specifying the first pin and, indirectly, the group of other required pins. See explanation below. All I/O pins will be set to output mode.
- ***Command*** is a variable/constant/expression (0 – 255) indicating an LCD command to send.
- ***OutputData*** is a list of variables, constants, expressions and formatters that tells LCDOUT how to format outgoing data. LCDOUT can transmit individual or repeating bytes, convert values into decimal, hex or binary text representations, or transmit strings of bytes from variable arrays. These actions can be combined in any order in the *OutputData* list.

Quick Facts

Table 5.49: LCDOUT Quick Facts.

	BS2p, BS2pe, and BS2px
Values for *Pin*	0, 1, 8 or 9
I/O Pin Arrangement when *Pin* is 0 or 1	0 or 1 (depending on pin): LCD Enable (E) pin 2: LCD Read/Write (R/W) pin 3: LCD Register Select (RS) pin 4 – 7: LCD Data Buss (DB4 – DB7, respectively) pins
I/O Pin Arrangement when *Pin* is 8 or 9	8 or 9 (depending on pin): LCD Enable (E) pin 10: LCD Read/Write (R/W) pin 11: LCD Register Select (RS) pin 12 – 15: LCD Data Buss (DB4 – DB7, respectively) pins
Special Notes	LCDOUT is designed to use the LCD's 4-bit mode only.
Related Commands	LCDCMD and LCDIN

Explanation

NOTE: LCDCMD, LCDIN and LCDOUT use a 4-bit interface to the LCD which requires a specific initialization sequence before LCDIN and LCDOUT can be used; read more below.

The three LCD commands (LCDCMD, LCDIN and LCDOUT) allow the BS2p, BS2pe, and BS2px to interface directly to standard LCD displays that feature a Hitachi 44780 controller (part #HD44780A). This includes many 1 x 16, 2 x 16 and 4 x 20 character LCD displays.

variable resistor affects the time it takes to discharge the capacitor from 5 volts to approximately 1.4 volts.

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 amount by which the internal value must be scaled varies with the size of the resistor being used.

Finding the best Scale value:
1. Build the circuit shown in Figure 5.27 and plug the BS1 into the PC.
2. In the BASIC Stamp editor select *Pot Scaling* from the *Run* menu. A special calibration window appears, allowing you to find the best value.
3. The window asks for the number of the I/O pin to which the variable resistor is connected. Select the appropriate pin (0-7) from the drop-down.
4. The editor downloads a short program to the BS1 (this overwrites any program already stored in the BS1).
5. The window will now show the Scale Factor. Adjust the resistor until the smallest number is shown for scale (assuming you can adjust the resistor, as with a potentiometer).
6. Once you've found the smallest number for scale, you're done. This number should be used for the Scale in the POT command.
7. Optionally, you can verify the scale number found above by selecting the POT Value checkbox (so it's checked). This locks the scale and causes the BS1 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, deselect the POT Value checkbox. Continue this process until you find the best scale.

Demo Program (POT.bs1)

```
' POT.bs1
' This program demonstrates the use of the POT command.  Connect one side
' of a 5-50K potentiometer to P0.  To the other side of the potentiometer
' connect a 0.1 uF capacitor, and then connect the other side of the
' capacitor to Vss (ground).  Before running demo program,
' use the Run | POT Scaling dialog to determine the best Scale factor.
```

```
  DEBUG "Program Slot #", DEC value.NIB0, CR

Main:
  FOR idx = 0 TO 9
    value = (idx + 3) * 8
    PUT idx, value
    DEBUG "  Writing: ", DEC2 value, " to location: ", DEC2 idx, CR
  NEXT
  DEBUG CR
  RUN 1
  END
```

Demo Program (GET_PUT2.bsx)

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

```
' GET_PUT2.bsx
' This example demonstrates the use of the GET and PUT commands.  First,
' the Slot location is read using GET to display the currently running
' program number.  Then a set of values are read (GET) from locations
' 0 to 9 and displayed on the screen for verification.  This program is a
' BS2SX project consisting of GET_PUT1.BSX and GET_PUT2.BSX, but will run
' on the BS2e, BS2p, BS2pe, and BS2px without modification.

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

#SELECT $STAMP
  #CASE BS2
    #ERROR "BS2e or greater required."
  #CASE BS2E, BS2SX
    Slot        CON     63
  #CASE BS2P, BS2PE, BS2PX
    Slot        CON     127
#ENDSELECT

value           VAR     Byte
idx             VAR     Byte


Setup:
  GET Slot, value
  DEBUG "Program Slot #", DEC value.NIB0, CR

Main:
  FOR idx = 0 TO 9
    GET idx, value
    DEBUG "  Reading: ", DEC2 value, " from location: ", DEC2 idx, CR
  NEXT
  END
```

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.

When the Slot 1 program runs you may be surprised to see that cats and dogs are now zero and fleas are up to 259! – even though we didn't explicitly define them. What happened? The key to remember is that variable names are simply pointers to RAM addresses, and the PBASIC compiler assigns variable names to RAM in descending order by size. This means that in the Slot 1 program, fleas was assigned to RAM locations 0 and 1 which are holding the values 3 and 1 respectively. Since words are stored low-byte first, the value 259 for fleas makes sense (3 + (1 * 256)).

There may be occasions when you need to preserve the RAM space in a program slot before calling on another slot that has different variable requirements. You can use the following subroutines to save your RAM space to the SPRAM and restore it on returning from the other program slot.

```
Save_RAM:
  PUT 0, B0                             ' move RAM 0 value to SP
  FOR B0 = 1 TO 25                      ' loop through other RAM bytes
    PUT B0, B0(B0)                      ' move RAM value to SP location
  NEXT
  RETURN

Restore_RAM:
  FOR B0 = 1 TO 25                      ' loop through RAM
    GET B0, B0(B0)                      ' retrieve RAM value from SP
  NEXT
  GET 0, B0                             ' retrieve RAM 0 value from SP
  RETURN
```

While the use of internal variable names is usually discouraged, these subroutines demonstrate a valid opportunity for their use, as well as the ability to take advantage of the BASIC Stamp's unique memory architecture.

The Save_RAM routine starts by saving the first byte of RAM (internal name: B0) to location 0 in the SPRAM. This is done so that B0 can be used as a loop index for the other locations. The FOR...NEXT loop provides control of that index. The following line is probably the most difficult to comprehend, but works due to the nature of the BASIC Stamp module's RAM organization

```
  PUT B0, B0(B0)                      ' move RAM value to SP location
```

ADDITIONAL CONVERSION FORMATTERS.

All 2

The decimal formatter is only one of a whole family of conversion formatters available with SERIN on all the BS2 models. See Table 5.100 for a list of available conversion formatters. All of the conversion formatters work similar to the decimal formatter (as described in the "Decimal Formatter Specifics" section, above). The formatters receive bytes of data, waiting for the first byte that falls within the range of characters they accept (e.g., "0" or "1" for binary, "0" to "9" for decimal, "0" to "9" and "A" to "F" for hex, and "-" for signed variations of any type). Once they receive a numeric character, they keep accepting input until a non-numeric character arrives or (in the case of the fixed length formatters) the maximum specified number of digits arrives.

Table 5.107: 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

NOTE: For "open" baudmodes used in networking, add 32768 to the values from the table above. If the dedicated serial port (*Tpin*=16) is used, the data is inverted and driven regardless of the baudmode setting.

Table 5.108: 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

NOTE: For "open" baudmodes used in networking, add 32768 to the values from the table above. If the dedicated serial port (*Tpin*=16) is used, the data is inverted and driven regardless of the baudmode setting.

Table 5.109: 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. See the SERIN command description for more information.

A SIMPLE FORM OF SEROUT.

The example below will transmit a single byte through I/O pin 1 at 2400 baud, 8N1, inverted:

```
SEROUT 1, N2400, (65)
```

--or--

```
SEROUT 1, 16780, [65]
```

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

a. Unmatched settings on the sender and receiver side will cause garbled data transfers or no data transfers. If the data you receive is unreadable, it is most likely a baud rate setting error.

5. If data transmitted to the Stamp Editor's Debug Terminal is garbled, verify the output format.
   a. A common mistake is to send data with SEROUT in ASCII format. For example, SEROUT 16, 84, [0] instead of SEROUT 16, 84, [DEC 0]. The first example will send a byte equal to 0 to the PC, resulting in the Debug Terminal clearing the screen (since 0 is the control character for a clear-screen action).

Demo Program (SEROUT.bs1)

```
' SEROUT.bs1
' This program transmits the string "ABCD" followed by a number and a
' carriage-return at 2400 baud, inverted, N81 format.

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

SYMBOL  SOut            = 1
SYMBOL  Baud            = N2400

SYMBOL  value           = W1


Setup:
  value = 1

Main:
  SEROUT SOut, Baud, ("ABCD", #value)
  value = value + 1
  PAUSE 250
  GOTO Main
  END
```

Demo Program (SERIN_SEROUT1.bs2)

NOTE: This example program was written for BS2's 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_SEROUT1.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
' Sender. 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
```

To ensure accuracy of SLEEP intervals, the BASIC Stamp periodically compares the watchdog timer to the more-accurate resonator time base. It calculates a correction factor that it uses during SLEEP. As a result, longer SLEEP intervals are accurate to approximately ±1 percent.

If your application is driving loads (sourcing or sinking current through output-high or output-low pins) during SLEEP, current will be interrupted for about 18 ms (60 µs on the BS2pe) when the BASIC Stamp wakes up every 2.3 seconds. The reason is that the watchdog-timer reset that awakens the BASIC Stamp also causes all of the pins to switch to input mode for approximately 18 ms. When the interpreter firmware regains control of the processor, it restores the I/O directions dictated by your program.

If you plan to use END, NAP, POLLWAIT or SLEEP in your programs, make sure that your loads can tolerate these periodic power outages. The simplest solution is often to connect resistors high or low (to +5V or ground) as appropriate to ensure a continuing supply of current during the reset glitch. The demo program demonstrates the effects of this glitch.

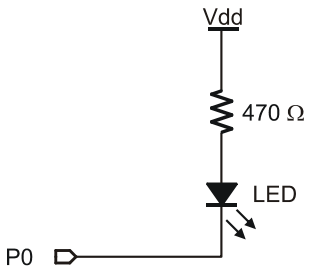


Figure 5.45: SLEEP Example LED Circuit.

Demo Program (SLEEP.bs2)

NOTE: This example program is written for the BS2, but it also can be used with the BS1 and all other BS2 models by changing the $STAMP directive accordingly.

```
' SLEEP.bs2
' This program lights an LED and then goes to sleep. Connect an LED to pin
' 0 as shown in the description of SLEEP in the manual and run the program.
' The LED will turn on, then the BASIC Stamp will go to sleep. During
' sleep,the LED will remain on, but will blink at intervals of
' approximately 2.3 seconds due to the watchdog timeout and reset.

' {$STAMP BS2}

Setup:
  LOW 0                                 ' turn LED on
```

STOP

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

All 2

STOP

Function

Stop program execution.

Quick Facts

Table 5.120: STOP Quick Facts.

	All BS2 Models
Related Command	END

Explanation

STOP prevents the BASIC Stamp from executing any further instructions until it is reset. The following actions will reset the BASIC Stamp:

1. Pressing and releasing the RESET button on the development board.
2. Driving the RES pin low then letting it float (high).
3. Downloading a new program
4. Disconnecting then reconnecting the power.

STOP differs from END in two respects:

1. Stop does not put the BASIC Stamp into low-power mode. The BASIC Stamp draws just as much current as if it were actively running program instructions.
2. The output glitch that occurs after a program has "ended" does not occur after a program has "stopped."

All 2

Demo Program (STOP.bs2)

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

```
' STOP.bs2
' This program is similar to SLEEP.BS2 except that the LED will not blink
' since the BASIC Stamp does not go into low power mode.  Use the circuit
' shown in the description of the SLEEP command for this example.

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


Main:
  LOW 0                                 ' turn LED on
  STOP                                  ' stop program
```

The dim/bright commands support 19 brightness levels. Lamp modules may also be turned on and off using the standard UnitOn and UnitOff commands. In the example instruction above, we dimmed the lamp by first turning it completely off, then sending 10 cycles of the Dim command. This may seem odd, but it follows the peculiar logic of the X-10 system.

Demo Program (X10.bs2)

All 2

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

```
' XOUT.BS2
' This program--really two program fragments--demonstrates the syntax and
' use of the XOUT command. XOUT works like pressing the buttons on an X-10
' control box; first you press one of 16 keys to identify the unit you want
' to control, then you press the key for the action you want that unit to
' take (turn ON, OFF, Bright, or Dim). There are also two group-action
' keys, Lights ON and All OFF. Lights ON turns all lamp modules on without
' affecting appliance modules. All OFF turns off all modules, both lamp and
' appliance types.  Connect the BASIC Stamp to a power-line interface as
' shown in the XOUT command description in the manual.

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

Mpin            PIN     0               ' modulation pin
Zpin            PIN     1               ' zero-cross input

HouseA          CON     0               ' House code A = 0
Unit1           CON     0               ' Unit code 1 = 0
Unit2           CON     1               ' Unit code 2 = 1

' This first example turns a standard (appliance or non-dimmer lamp) module
' ON, then OFF. Note that once the Unit code is sent, it need not be
' repeated
' --subsequent instructions are understood to be addressed to that unit.

Main:
  XOUT Mpin, Zpin, [HouseA\Unit1\2]     ' select Unit1 (appliance module)
  XOUT Mpin, Zpin, [HouseA\UNITON]      ' turn it on

  PAUSE 1000                            ' wait one second

  XOUT Mpin, Zpin, [HouseA\UNITOFF]     ' then turn it off

' The next example talks to a lamp module using the dimmer feature. Dimmers
' go from full ON to dimmed OFF in 19 steps. Because dimming is relative to
' the current state of the lamp, the only guaranteed way to set a
' predefined brightness level is to turn the dimmer fully OFF, then ON,
' then dim to the desired level.

  XOUT Mpin, Zpin, [HouseA\Unit2\2]     ' select Unit2 (lamp module)
```