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-p

BASIC Stamp 1

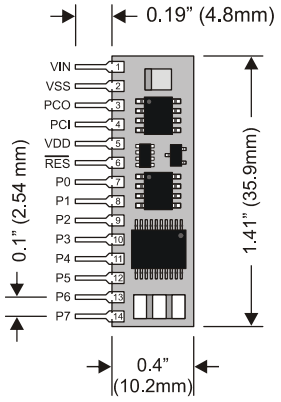

Figure 1.1: BASIC Stamp 1 (Rev B) (Stock# BS1-IC).

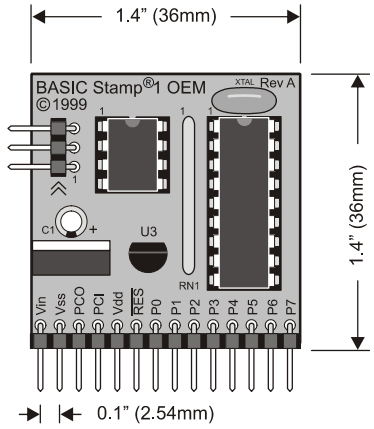

Figure 1.2: BASIC Stamp 1 OEM (Rev. A) (Stock# 27295).

BASIC Stamp 2e

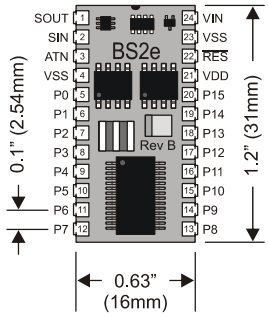

Figure 1.6: BASIC Stamp 2e (Rev. B) (Stock# BS2E-IC).

The BASIC Stamp 2e is available in the above 24-pin DIP package.

Table 1.3: BASIC Stamp 2e Pin Descriptions.

Pin	Name	Description
1	SOUT	Serial Out: connects to PC serial port RX pin (DB9 pin 2 / DB25 pin 3) for programming.
2	SIN	Serial In: connects to PC serial port TX pin (DB9 pin 3 / DB25 pin 2) for programming.
3	ATN	Attention: connects to PC serial port DTR pin (DB9 pin 4 / DB25 pin 20) for programming.
4	VSS	System ground: (same as pin 23) connects to PC serial port GND pin (DB9 pin 5 / DB25 pin 7) for programming.
5-20	P0-P15	General-purpose I/O pins: each can source and sink 30 mA. However, the total of all pins should not exceed 75 mA (source or sink) if using the internal 5-volt regulator. The total per 8-pin groups (P0 – P7 or P8 – 15) should not exceed 100 mA (source or sink) if using an external 5-volt regulator.
21	VDD	5-volt DC input/output: if an unregulated voltage is applied to the VIN pin, then this pin will output 5 volts. If no voltage is applied to the VIN pin, then a regulated voltage between 4.5V and 5.5V should be applied to this pin.
22	RES	Reset input/output: goes low when power supply is less than approximately 4.2 volts, causing the BASIC Stamp to reset. Can be driven low to force a reset. This pin is internally pulled high and may be left disconnected if not needed. Do not drive high.
23	VSS	System ground: (same as pin 4) connects to power supply's ground (GND) terminal.
24	VIN	Unregulated power in: accepts 5.5 - 12 VDC (7.5 recommended), which is then internally regulated to 5 volts. Must be left unconnected if 5 volts is applied to the VDD (+5V) pin.

the variable RAM for these models, only the BS2p40 module has the extra 16 I/O pins for which this feature is intended.

THE INPUT/OUTPUT VARIABLES.

The word variable INS is unique in that it is read-only. The 16 bits of INS reflect the state of I/O pins P0 through P15. It may only be read, not written. OUTS contains the states of the 16 output latches. DIRS controls the direction (input or output) of each of the 16 I/O pins.

A 0 in a particular DIRS bit makes the corresponding pin an input and a 1 makes the corresponding pin an output. So if bit 5 of DIRS is 0 and bit 6 of DIRS is 1, then I/O pin 5 (P5) is an input and I/O pin 6 (P6) is an output. A pin that is an input is at the mercy of circuitry outside the BASIC Stamp; the BASIC Stamp cannot change its state. A pin that is an output is set to the state indicated by the corresponding bit of the OUTS register.

When the BASIC Stamp is powered up, or reset, all memory locations are cleared to 0, so all pins are inputs (DIRS = %0000000000000000). Also, if the PBASIC program sets all the I/O pins to outputs (DIRS = %1111111111111111), then they will initially output low, since the output latch (OUTS) is cleared to all zeros upon power-up or reset, as well.

Table 4.2: RAM Organization for all BS2 models.

NOTE: There are 16 words, of two bytes each for a total of 32 bytes*. All bits are individually addressable through variable modifiers; the bits within the upper three words are also individually addressable though the pre-defined names shown. All registers are word, byte, nibble and bit addressable.

*The BS2p, BS2pe, and BS2px have an additional set of INS, OUTS, and DIRS registers that are switched in and out of the memory map in place of the main INS, OUTS, and DIRS registers by using AUXIO, MAINIO, and IOTERM. Only the BS2p40 has the required extra I/O pins this feature is intended for.

Word Name	Byte Names	Nibble Names	Bit Names	Special Notes
INS*	INL, INH	INA, INB INC, IND	IN0 – IN7 IN8 – IN15	Input pins
OUTS*	OUTL, OUTH	OUTA, OUTB OUTC, OUTD	OUT0 – OUT7 OUT8 – OUT15	Output pins
DIRS*	DIRL, DIRH	DIRA, DIRB DIRC, DIRD	DIR0 – DIR7 DIR8 – DIR15	I/O pin direction control
W0	B0, B1			
W1	B2, B3			
W2	B4, B5			
W3	B6, B7			
W4	B8, B9			
W5	B10, B11			
W6	B12, B13			
W7	B14, B15			
W8	B16, B17			
W9	B18, B19			
W10	B20, B21			
W11	B22, B23			
W12	B24, B25			

BRANCH

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

1 **BRANCH** ***Offset*****, (*****Address0, Address1, …AddressN*** **)**

All 2 **BRANCH** ***Offset*****, [*****Address0, Address1, …AddressN*** **]**

Function

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

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

- ***Offset*** is a variable/constant/expression (0 – 255) that specifies the index of the address, in the list, to branch to (0 – N).
- ***Addresses*** are labels that specify where to go. BRANCH will ignore any list entries beyond offset 255.

Quick Facts

Table 5.3: BRANCH Quick Facts.

	BS1	All BS2 Models
Limit of *Address* Entries	Limited only by memory	256
Related Commands	None	ON...GOTO

Explanation

The BRANCH instruction is useful when you want to write something like this:

```
IF value = 0 THEN Case_0              ' when value is 0, jump to Case_0
IF value = 1 THEN Case_1              ' when value is 1, jump to Case_1
IF value = 2 THEN Case_2              ' when value is 2, jump to Case_2
```

1 BS1 syntax not shown here.

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

```
BRANCH value, [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 *value* is greater than 2), BRANCH does nothing and 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.

Every high bit (1) in the *PinMask* argument enables the output direction for the corresponding I/O pin while every low bit (0) disables the output direction. In the above example, I/O pins 8, 4, 1, and 0 are set to the output direction and all other I/O pins are set to the input direction. This is similar to the following statement:

```
DIRS = %0000000100010011
```

PULL-UP RESISTORS.

Pull-up resistors are commonly used in circuitry where a component, such as a button, provides an open/drain signal; the signal is either floating (open) or is driven to ground (drain). Since the BASIC Stamp input pins must always be connected to either 5 volts or ground (0 volts) in order to read a reliable logic state with them, a pull-up resistor is required on circuitry, such as the button circuit mentioned above, so that the signal is never left floating (electrically disconnected).

The following example enables internal pull-up resistors on I/O pins 15, 12, 6, and 3, and disables internal pull-up resistors on all other I/O pins:

```
CONFIGPIN  PULLUP, %1001000001001000
```

Note that the internal pull-up resistors are intentionally weak, about 20 kΩ. Additionally, the internal pull-up resistors can be activated for all pins, regardless of pin direction, but really matter only when the associated pin is set to input mode.

LOGIC THRESHOLD.

An input pin's logic threshold determines the voltage levels that are interpreted as logic high (1) and logic low (0). Most microcontrollers, and other integrated circuits use one of two types of logic threshold: TTL Level or CMOS Level. The BASIC Stamp I/O pins are, by default, configured for TTL level logic thresholds. Figure 5.2 is an illustration of the difference between TTL and CMOS logic levels.

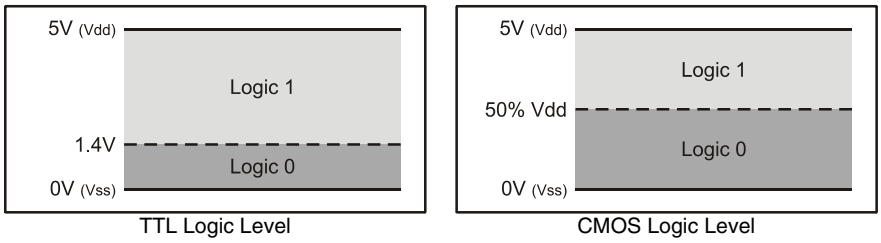

Figure 5.2: TTL and CMOS Logic Level Threshold Voltages

DEBUG

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

1 All 2

DEBUG ***OutputData*** **{ , *OutputData* }**

Function

Display information on the PC screen within the BASIC Stamp Editor's Debug Terminal. This command can be used to display text or numbers in various formats on the PC screen in order to follow program flow (called debugging) or as part of the functionality of the BASIC Stamp application.

1

NOTE: Expressions are not allowed as arguments on the BS1. The only constant allowed for the BS1 DEBUG command is a text string.

- ***OutputData*** is a variable/constant/expression (0 – 65535) that specifies the information to output. Valid data can be ASCII characters (text strings and control characters), decimal numbers (0 - 65535), hexadecimal numbers ($0000 - $FFFF) or binary numbers (up to %1111111111111111). Data can be modified with special formatters as explained below.

Quick Facts

Table 5.9: DEBUG Quick Facts.

	BS1	BS2, BS2e, BS2sx BS2p, BS2pe	BS2px
Serial Protocol	Asynchronous 4800, N, 8, 1 True polarity Custom packetized format	Asynchronous 9600, N, 8, 1 Inverted polarity Raw data	Asynchronous 19200, N, 8, 1 Inverted polarity Raw data
Related Commands	None	SEROUT and DEBUGIN	

Explanation

DEBUG provides a convenient way for your BASIC Stamp to send messages to the PC screen while running. The name "debug" suggests its most popular use; debugging programs by showing you the value of a variable or expression, or by indicating what portion of a program is currently executing. DEBUG is also a great way to rehearse programming techniques. Throughout this manual, we use DEBUG to give you immediate feedback on the effects of instructions. The following example demonstrates using the DEBUG command to send the text string message "Hello World!".

```
DEBUG  "Hello, World!"
```

After you download this one-line program, the BASIC Stamp Editor will open a Debug Terminal on your PC screen and wait for a response from

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
```

DEBUGIN

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

All 2

DEBUGIN ***InputData*** **{ , *InputData* }**

NOTE: DEBUGIN requires the $PBASIC 2.5 compiler directive.

Function

Accept information from the user via the Debug Terminal within the BASIC Stamp Editor program.

- ***InputData*** is list of variables and formatters that tells DEBUGIN what to do with incoming data. DEBUGIN can store data in a variable or array, interpret numeric text (decimal, binary, or hex) and store the corresponding value in a variable, wait for a fixed or variable sequence of bytes, or ignore a specified number of bytes. These actions can be combined in any order in the *InputData* list.

Quick Facts

Table 5.14: DEBUGIN Quick Facts.

	BS2, BS2e, BS2sx, BS2p, BS2pe	BS2px
Serial Protocol	Asynchronous 9600 baud N, 8, 1 Inverted Polarity, Raw Data	Asynchronous 19200 baud N, 8, 1 Inverted Polarity, Raw Data
Related Commands	SERIN and DEBUG	

Explanation

DEBUGIN provides a convenient way for your BASIC Stamp to accept input from the user via the Debug Terminal. DEBUGIN can wait for, filter and convert incoming data in powerful ways, using the same techniques and modifiers as SERIN.

DEBUGIN is actually a special case of the SERIN instruction. It is set for inverted (RS-232-compatible) serial input through the programming connector (the SIN pin) at 9600 baud (19200 baud on BS2px), no parity, 8 data bits, and 1 stop bit.

For example:

```
DEBUGIN DEC1 myNum
```

FOR…NEXT

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

1

FOR ***Counter*** **=** ***StartValue*** **TO** ***EndValue*** **{ STEP {-}** ***StepValue*** **}**
Statement(s)
NEXT { *Counter* }

All 2

FOR ***Counter*** **=** ***StartValue*** **TO** ***EndValue*** **{ STEP** ***StepValue*** **}**
Statement(s)
NEXT { *Counter* }

Function

Create a repeating loop that executes the *Statement(s)*, one or more program lines that form a code block, between FOR and NEXT, incrementing or decrementing *Counter* according to *StepValue* until the value of the *Counter* variable passes the *EndValue*.

1

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

- ***Counter*** is a variable (usually a byte or a word) used as a counter.
- ***StartValue*** is a variable/constant/expression (0 – 65535) that specifies the initial value of the variable (*Counter*).
- ***EndValue*** is a variable/constant/expression (0 – 65535) that specifies the end value of the variable (*Counter*). When the value of *Counter* is outside of the range *StartValue* to *EndValue*, the FOR...NEXT loop stops executing and the program goes on to the instruction after NEXT.

1

NOTE: Use a minus sign to indicate negative *StepValues* on the BS1.

- ***StepValue*** is an optional variable/constant/expression (0 – 65535) by which the *Counter* increases or decreases with each iteration through the FOR…NEXT loop. On the BS1, use a minus sign (-) in front of the *StepValue* to indicate a negative step. On all BS2 models, if *StartValue* is larger than *EndValue*, PBASIC understands *StepValue* to be negative, even though no minus sign is used.
- ***Statement*** is any valid PBASIC instruction.

pin 0 is set low, the BASIC Stamp will set I/O pin 1 high. It will continue to perform this operation, in-between each command in the loop, endlessly.

THE BASIC STAMP "REMEMBERS" THE POLLING CONFIGURATION FOR THE DURATION OF THE PBASIC PROGRAM.

It's important to note that, in this example, only the DEBUG and GOTO commands are being executed over and over again. The first three lines of code are only run once, yet their effects are "remembered" by the BASIC Stamp throughout the rest of the program.

FOR COMPARISON: ACHIEVING THE SAME EFFECTS WITHOUT THE POLLING COMMANDS.

If the polling commands were not used, the program would have to look like the one below in order to achieve the same effect.

```
INPUT 0
OUTPUT 1

Main:
  OUT1 = ~IN0
  DEBUG "Looping...", CR
  OUT1 = ~IN0
  GOTO Main
```

In this example, we create the inverse relationship of input pin 0 and output pin 1 manually, in-between the DEBUG and GOTO lines. Though the effects are the same as when using the polling commands, this program actually takes a little longer to run and consumes 7 additional bytes of program (EEPROM) space. Clearly, using the polling commands is more efficient.

USING MULTIPLE POLLED-INPUT AND POLLED-OUTPUT PINS.

You can have as many polled-input and polled-output pins as you have available. If multiple polled-input pins are defined, any one of them can trigger changes on the polled-output pins that are also defined. For example:

```
POLLIN 0, 0
POLLIN 1, 0
POLLOUT 2, 1
POLLOUT 3, 1
POLLMODE 2

Main:
  DEBUG "Looping...", CR
  GOTO Main
```

This code sets I/O pins 0 and 1 to polled-input pins (looking for a low (0) state) and sets I/O pins 2 and 3 to polled-output pins (with a high-active

```
INPUT 0
OUTPUT 1

Main:
  OUT1 = ~IN0
  DEBUG "Looping...", CR
  OUT1 = ~IN0
  GOTO Main
```

In this example, we create the inverse relationship of input pin 0 and output pin 1 manually, in-between the DEBUG and GOTO lines. Though the effects are the same as when using the polling commands, this program actually takes a little longer to run and consumes 7 additional bytes of program (EEPROM) space. Clearly, using the polling commands is more efficient.

USING MULTIPLE POLLED-INPUT AND POLLED-OUTPUT PINS.

You can have as many polled-input and polled-output pins as you have available. If multiple polled-output pins are defined, all of them change in response to changes on the polled-input pins. For example:

```
POLLIN 0, 0
POLLOUT 1, 0
POLLOUT 2, 1
POLLOUT 3, 1
POLLMODE 2

Main:
  DEBUG "Looping...", CR
  GOTO Main
```

This code sets up I/O pin 0 as a polled-input pin (looking for a low (0) state) and sets I/O pins 1, 2 and 3 to polled-output pins. Polled-output pin 1 is set to a low-active state and pins 2 and 3 are set to a high-active state. If I/O pin 0 goes low, the BASIC Stamp will set I/O pin 1 low and I/O pins 2 and 3 high. The table below shows the two possible states of the polled-input pin and the corresponding states the BASIC Stamp will set the polled-output pins to.

Table 5.79: POLLOUT Truth Table.

Polled-Input	Polled-Outputs		
0	**1**	**2**	**3**
1	1	0	0
0	0	1	1

POLLED-OUTPUTS CAN BE "LATCHED" ALSO.

Normally, any polled-output pins reflect the state changes continuously, as described above. The POLLMODE command supports another feature,

POLLRUN – BASIC Stamp Command Reference

A SIMPLE POLLRUN EXAMPLE.

The following is a simple example of the POLLRUN command.

```
POLLIN 0, 0
POLLRUN 1
POLLMODE 3

Main:
  DEBUG "Waiting in Program Slot 0...", CR
  GOTO Main
```

The first line of the above code will set up I/O pin 0 as a polled-input pin looking for a low (0) state. The second line, POLLRUN, tells the BASIC Stamp that when I/O pin 0 goes low, it should switch execution over to the program residing in program slot 1. The third line, POLLMODE, activates the polled-run configuration.

Once the BASIC Stamp reaches the *Main* routine, it will continuously print "Waiting in Program Slot 0…" on the PC screen. In between reading the DEBUG and GOTO commands, however, the BASIC Stamp will poll I/O pin 0 and check for a high or low state. If the state of pin 0 is high, it will do nothing and continue as normal. If the state of pin 0 is low, it will switch execution over to the program in slot 1 (the second program is not shown in this example). The switch to another program slot works exactly like with the RUN command; the designated program is run and the BASIC Stamp does not "return" to the previous program (similar to a GOTO command).

Note that in order for the polled-run activity to occur, the poll mode must be set to either 3 or 4 (the two modes that activate polled-run). Also note, that the polled-run modes, 3 and 4, are unique. As soon as the polled-run action occurs, the mode switches to 1 (deactivated, saved) or 2 (activated, outputs), respectively. This is so that the BASIC Stamp doesn't continuously go to the start of the designated program slot while the polled-inputs are in the desired poll state. Without this "one shot" feature, your program would appear to lock-up as long as the polled-inputs are in the designated state.

After the program switch takes place, the *ProgramSlot* value is maintained. Any future change to poll mode 3 or 4, without another POLLRUN command, will result in the previously defined program slot being used.

```
Main:
  PULSIN Pulse, 1, time                 ' measure positive pulse
  IF time = 0 THEN Main                 ' if 0, try again
  DEBUG CLS, time                       '   else display result
  GOTO Main
  END
```

All 2

Demo Program (PULSIN.bs2)

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

```
' PULSIN.bs2
' This program uses PULSIN to measure a pulse generated by discharging a
' 0.1 uF capacitor through a 1K resistor. Pressing the switch generates
' the pulse, which should ideally be approximately 120 us (60 PULSIN units
' of 2 us) long (for BS2 and BS2e). Variations in component values may
' produce results that are up to 10 units off from this value.  For more
' information on calculating resistor-capacitor timing, see the RCTIME
' command.

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

Pulse           PIN     7               ' pulse input pin

#SELECT $STAMP
  #CASE BS2, BS2E, BS2PE
    Scale       CON     $200            ' 2.0 us per unit
  #CASE BS2SX, BS2P
    Scale       CON     $0CC            ' 0.8 us per unit
  #CASE BS2PX
    Scale       CON     $0CF            ' 0.81 us per unit

#ENDSELECT

time            VAR     Word

Main:
  PULSIN Pulse, 1, time                 ' measure positive pulse
  IF (time > 0) THEN                    ' if not 0
    DEBUG HOME,
          DEC time, " units ", CLREOL   ' display raw input
    time = time */ Scale                ' adjust for Stamp
    DEBUG CR,
          DEC time, " us  "             ' display microseconds
  ELSE
    DEBUG CLS, "Out of Range"           ' else error message
  ENDIF
  PAUSE 200
  GOTO Main
  END
```

```
        DEC Result, CR
  HIGH Coil                          ' release relay
  PAUSE 1000                         ' wait one second
LOOP
END
```

Figure 5.36 shows the pinouts of the two styles of PC serial ports and how to connect them to the BASIC Stamp's I/O pin (the 22 kΩ resistor is not needed if connecting to the SIN pin). Though not normally needed, the figure also shows loop back connections that defeat hardware handshaking used by some PC software. Note that PC serial ports are always male connectors. The 25-pin style of serial port (called a DB25) looks similar to a printer (parallel) port except that it is male, whereas a parallel port is female.

SERIAL TIMING AND MODE (BAUDMODE).

Asynchronous serial communication relies on precise timing. Both the sender and receiver must be set for identical timing, usually expressed in bits per second (bps) called baud.

On all BASIC Stamp models, SERIN requires a value called *Baudmode* that tells it the important characteristics of the incoming serial data; the bit period, number of data and parity bits, and polarity.

On the BS1, serial communication is limited to: no-parity, 8-data bits and 1-stop bit at one of four different speeds: 300, 600, 1200 or 2400 baud. Table 5.95 indicates the *Baudmode* value or symbols to use when selecting the desired mode.

Table 5.95: BS1 Baudmode Values.

Baudmode Value	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

On all BS2 models, serial communication is very flexible. The *Baudmode* argument for SERIN accepts a 16-bit value that determines its characteristics: 1-stop bit, 8-data bits/no-parity or 7-data bits/even-parity and virtually any speed from as low as 300 baud to greater than 100K baud (depending on the BASIC Stamp). Table 5.96 shows how *Baudmode* is calculated, while Table 5.97, Table 5.98, and Table 5.99 show common baud modes for standard serial baud rates.

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 7-bit/even-parity	= =	0 8192
Step 3: Select polarity (bit 14).	True (noninverted) Inverted	= =	0 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

SHIFTOUT – BASIC Stamp Command Reference

SLEEP

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

1 All 2

SLEEP *Duration*

Function

1

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

Put the BASIC Stamp into low-power mode for a specified time.

- ***Duration*** is a variable/constant/expression (1 – 65535) that specifies the duration of sleep. The unit of time for *Duration* is 1 second, though the BASIC Stamp rounds up to the nearest multiple of 2.3 seconds.

Quick Facts

Table 5.118: SLEEP Quick Facts.

NOTE: Current measurements are based on 5-volt power, no extra loads and 75° F ambient temperature.

	BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px
Current Draw during Run	1 mA	3 mA	25 mA	60 mA	40 mA	15 mA	55 mA
Current Draw during SLEEP	25 µA	50 µA	200 µA	500 µA	350 µA	36 µA	450 µA
Related Commands	END and NAP				END, NAP and POLLWAIT		
Accuracy of SLEEP	±1% @ 75°F with stable power supply						

Explanation

SLEEP allows the BASIC Stamp to turn itself off, then turn back on after a programmed period of time. The length of SLEEP can range from 2.3 seconds to slightly over 18 hours. Power consumption is reduced to the amount described in Table 5.118, assuming no loads are being driven. The resolution of the SLEEP instruction is 2.304 seconds. SLEEP rounds the specified number of seconds up to the nearest multiple of 2.304. For example, SLEEP 1 causes 2.304 seconds of sleep, while SLEEP 10 causes 11.52 seconds (5 x 2.304) of sleep.

Pins retain their previous I/O directions during SLEEP. However, outputs are interrupted every 2.3 seconds during SLEEP due to the way the chip keeps time. The alarm clock that wakes the BASIC Stamp up is called the watchdog timer. The watchdog is a resistor/capacitor oscillator built into the interpreter chip. During SLEEP, the chip periodically wakes up and adjusts a counter to determine how long it has been asleep. If it isn't time to wake up, the chip "hits the snooze bar" and goes back to sleep.

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
```

WRITE – BASIC Stamp Command Reference