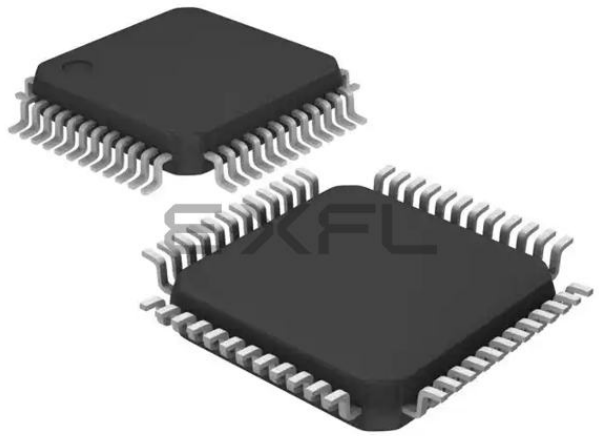


Welcome to E-XFL.COM

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	Surface Mount
Package / Case	48-LQFP
Supplier Device Package	48-TQFP
Purchase URL	https://www.e-xfl.com/product-detail/parallax/pbasic48w-px24

Preface

Book	Part #	Author and Publisher	Availability	
			PDF	In Print
What's a Microcontroller?	28123	Andy Lindsay; Parallax Inc.; ISBN 1-928982-02-6	Yes	Yes
Robotics with the Boe-Bot	28125	Andy Lindsay; Parallax Inc.; ISBN 1-928982-03-4	Yes	Yes
IR Remote for the Boe-Bot	70016	Andy Lindsay; Parallax Inc.; ISBN 1-928982-31-X	Yes	Yes
Basic Analog and Digital	28129	Andy Lindsay; Parallax Inc.; ISBN 1-928982-04-2	Yes	Yes
Applied Sensors	28127	Tracy Allen, PhD.; Parallax Inc.; ISBN 1-928982-21-2	Yes	Yes
Understanding Signals	28119 (With Full Kit)	Doug Pientak; Parallax Inc.; ISBN 1-928982-23-9	Yes	Yes
Industrial Control	27341	Marty Hebel / Will Devenport; Parallax Inc.; ISBN 1-928982-08-5	Yes	Yes
Elements of Digital Logic	70008	John Barrowman; Parallax Inc.; ISBN 1-928982-20-4	Yes	Yes
The Microcontroller Application Cookbook Volumes 1 and 2	Vol. 1&2: 28113 Vol. 2: 28112	Matt Gilliland; Woodglen Press; ISBN 0-616-11552-7 and 0-972-01590-6	No	Yes
Al's "World Famous" Stamp Project of the Month Anthology	70013	Al Williams; Parallax Inc.; ISBN 1-928982-25-5	Portions	Yes
The Nuts and Volts of BASIC Stamps Volumes 1, 2, 3, 4, and 5	Vol. 4: 70010 Vol. 5: 70015	Jon Williams, Scott Edwards and Lon Glazner; Parallax, Inc.; ISBN 1-928982-10-7, 1-928982-11-5, 1-928982-17-4, 1-928982-24-7 and 1-928982-30-1	Yes (all)	Yes (Vol 4 and Vol 5)
StampWorks	27220	Jon Williams; Parallax, Inc.; ISBN 1-928982-07-7	Yes	Yes
Stamp 2 Communication and Control Projects	70004	Thomas Petruzzellis; McGraw-Hill; ISBN 0-071411-97-6	No	Yes
Programming and Customizing the BASIC Stamp Computer	27956	Scott Edwards; McGraw-Hill; ISBN 0-071371-92-3	No	Yes
BASIC Stamp 2p	70001	Claus Kuehnel and Klaus Zahnert; Parallax, Inc.; ISBN 1-928982-19-0	Yes	No

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.

Table 1.4: BASIC Stamp 2sx Pin Descriptions

See the "BASIC Stamp Programming Connections" section on page 27 for more information on the required programming connections between the PC and the BASIC Stamp.

Guidelines and Precautions

When using the BASIC Stamp, or any IC chip, please follow the guidelines below.

1. **Be alert to static sensitive devices and static-prone situations.**
    a. The BASIC Stamp, like other IC's, can be damaged by static discharge that commonly occurs touching grounded surfaces or other conductors. Environmental conditions (humidity changes, wind, static prone surfaces, etc) play a major role in the presence of random static charges. It is always recommended to use grounding straps and anti-static or static dissipative mats when handling devices like the BASIC Stamp. If the items above are not available, be sure to touch a grounded surface after you have approached the work area and before you handle static sensitive devices.
2. **Verify that all power is off before connecting/disconnecting.**
    a. If power is connected to the BASIC Stamp or any device it is connected to while inserting or removing it from a circuit, damage to the BASIC Stamp or circuit could result.
3. **Verify BASIC Stamp orientation before connection to development boards and other circuits.**
    a. Like other IC's, the BASIC Stamp should be inserted in a specific orientation in relation to the development board or circuit. Powering the circuit with an IC connected backwards will likely damage the IC and/or other components in the circuit. Most IC's have some form of a "pin 1 indicator" as do most IC sockets. This indicator usually takes the form of a dot, a half-circle, or the number 1 placed at or near pin 1 of the device.

       The BS1-IC has a "1" and a half-circle indicator on the backside of the module. Additionally, Figure 1.1 above indicates the pin numbering and labels.

       All BS2 series modules have a half-circle indicator on the topside of the module (see Figure 1.13). This indicates that pin number one is the first pin counterclockwise from the notch. The socket that accepts this 24-pin module also

Introducing the BASIC Stamp Editor

This section describes the BASIC Stamp Editor for Windows version 2.2. This software supports all 7 BASIC Stamp modules available as of February 2005, and all 3 versions of the PBASIC programming language, PBASIC 1.0, PBASIC 2.0, and PBASIC 2.5.

The Programming Environment

The BASIC Stamp Windows Editor, shown in Figure 3.1, was designed to be easy to use and mostly intuitive. Those that are familiar with standard Windows software should feel comfortable using the BASIC Stamp Windows Editor.

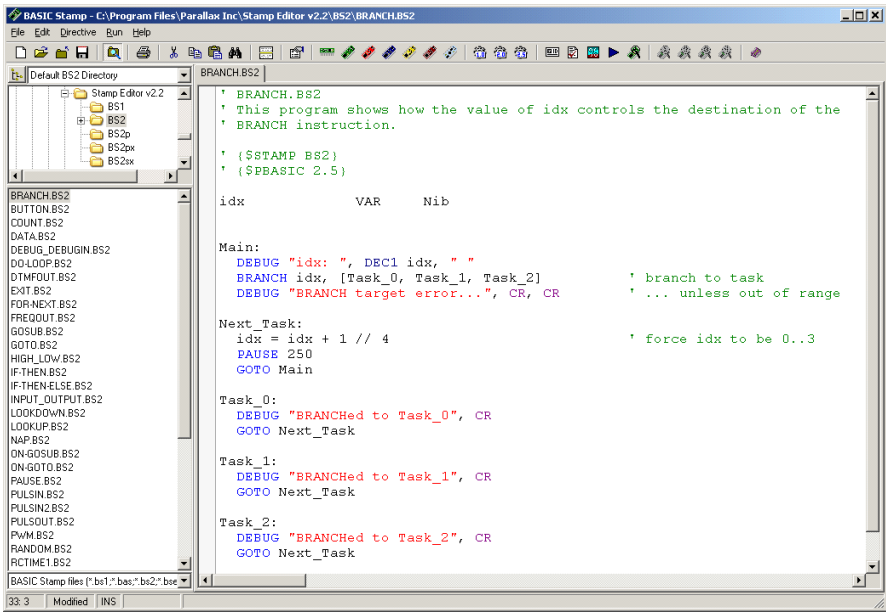


Figure 3.1: BASIC Stamp Windows Editor.

THE EDITOR WINDOW.

The editor window consists of the main edit pane with an integrated explorer panel to its left, as shown above.

THE MAIN EDIT PANE.

The main edit pane can be used to view and modify up to 16 different source code files at once. Each source code file that is loaded into the

make the Tab key move the cursor to positions, 3, 5, 7, 9 and 11, then afterwards, 13, 15, 17, 19, etc. (a multiple of two (11 – 9 = 2) after the last listed position.

FIXED PLUS SMART TABS.

The last option is a mixture of the first two, Fixed plus Smart Tabs (Figure 3.22); it is the default and recommended setting. Fixed plus Smart Tabs will cause the tab key to move the cursor to the position indicated by the Fixed Tab Positions field, or if the position is already beyond the end of that list, it reverts to Smart Tabs behavior. This setting, combined with a carefully configured Fixed Tab Positions field, allows for a fixed level of indenting on the left side of the source code (for executable code blocks), with very flexible indenting to the right of executable code (for comments that appear to the right of code). The default settings provide a quick, single-key method of indenting up to five (5) levels of executable code and easy alignment of multiple lines of comments to the right of that code.

Figure 3.20: Smart Tabs.

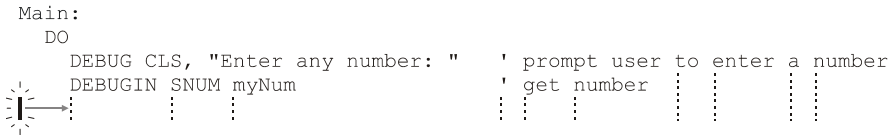


Figure 3.21: Fixed Tabs.

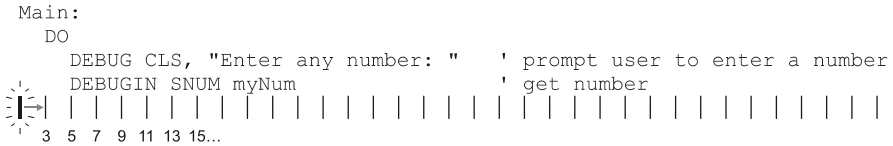
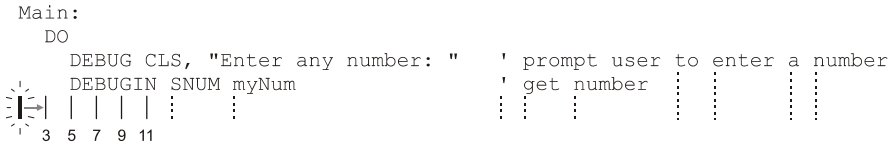


Figure 3.22: Fixed plus Smart Tabs.



DEBUG FUNCTION PREFERENCES.

Under the Debug Function tab (Figure 3.25), checkboxes allow enabling or disabling of special processing for 16 different control characters. The default is for all 16 control characters to be processed, but you may disable one or more of them if you are using the Debug Terminal to view data coming from a device other than a BASIC Stamp.

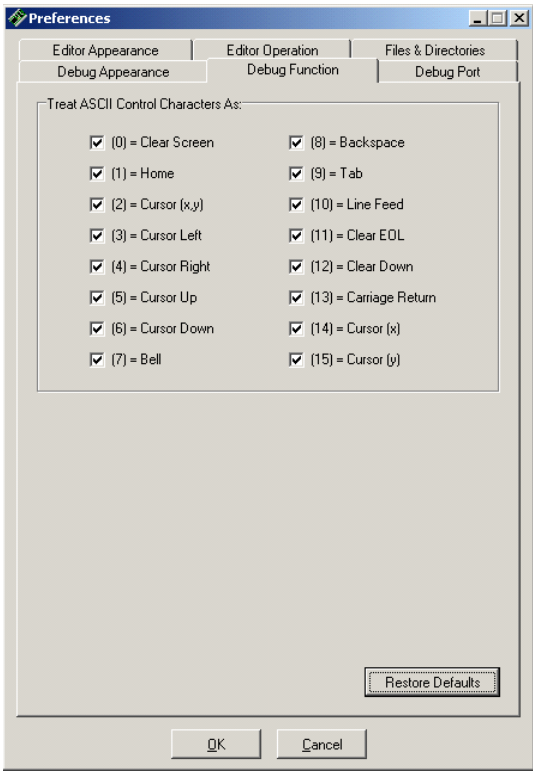


Figure 3.25: The Debug Function Tab under Edit → Preferences.

For example, a device that sends out a 0 to indicate something other than Clear Screen will cause unintentional clearing of the Receive pane; unchecking the checkbox for "(0) = Clear Screen" will prevent this from happening.

Advanced Compilation Techniques

INTRODUCTION TO BASIC STAMP PROJECTS.

For BS2e, BS2sx, BS2p, BS2pe and BS2px modules, each editor page can be a separate project, or part of a single project. A project is a set of up to eight files that should all be downloaded to the BASIC Stamp for a single application. Each of the files within the project is downloaded into a separate "program slot". Only the BASIC Stamp 2e, 2sx, 2p, 2pe, and 2px modules support multi-file projects.

USING THE $STAMP DIRECTIVE TO DEFINE MULTI-FILE PROJECTS.

For BASIC Stamp projects (consisting of multiple programs), the $STAMP directive has an option to specify additional filenames. The syntax below demonstrates this form of the $STAMP directive:

```
'  { $STAMP  BS2e, file2, file3, …, file8 }
```

Use this form of the $STAMP directive if a project, consisting of multiple files, is desired. This form of the directive must be entered only into the first program (to be downloaded into program slot 0). The *file2*, *file3*, etc. items should be the actual name (and optionally the path) of the other files in the project. *File2* refers to the program that should be downloaded into program slot 1, *file3* is the program that should be downloaded into program slot 2, etc. If no path is given, the filename is given the path of program 0 when loading them into the editor.

Up to seven filenames can be included, bringing the total to eight files in the project all together. Upon loading, tokenizing, running or viewing program 0 in the Memory Map, the editor will read the $STAMP directive, determine if the indicated files exist, will load them if necessary and change their captions to indicate the project they belong to and their associated program number. After the directive is tokenized properly, and all associated files are labeled properly, tokenizing, running or viewing any program in the Memory Map will result in that program's entire project being tokenized, downloaded or viewed.

When program #0 of a multi-file project is opened from diskette, the entire project will be loaded (all referenced files) as well. When a file that is part of a multi-file project is closed, the entire project (all the associated files) will be closed as well.

This feature is how the "string" capabilities of the DEBUG and SEROUT command expect to work. A string is simply a byte array used to store text. See "Displaying Strings (Byte Arrays)" in the DEBUG command description on page 166 for more information.

ALIASES AND VARIABLE MODIFIERS.

An *alias* is an alternative name for an existing variable. For example:


```
SYMBOL  cat             = B0            ' Create a byte-sized variable
SYMBOL  tabby           = cat           ' Create alias for cat
```

-- or --

```
cat             VAR     Byte            ' Create a byte-sized variable
tabby           VAR     cat             ' Create alias for cat
```

In this example, *tabby* is an alias to the variable *cat*. Anything stored in *cat* shows up in *tabby* and vice versa. Both names refer to the same physical piece of RAM. This kind of alias can be useful when you want to reuse a temporary variable in different places in your program, but also want the variable's name to reflect its function in each place. Use caution, because it is easy to forget about the aliases; during debugging, you might end up asking 'How did that value get here?!' The answer is that it was stored in the variable's alias.

On all the BS2 models, an alias can also serve as a window into a portion of another variable. This is done using "modifiers." Here the alias is assigned with a modifier that specifies what part to reference:

```
rhino           VAR     Word            ' A 16-bit variable
head            VAR     rhino.HIGHBYTE  ' Highest 8 bits of rhino
tail            VAR     rhino.LOWBYTE   ' Lowest 8 bits of rhino
```

Given that example, if you write the value %1011000011111101 to *rhino*, then *head* would contain %10110000 and *tail* would contain %11111101.

Table 4.3 lists all the variable modifiers. PBASIC 2.0 and 2.5 lets you apply these modifiers to any variable name and to combine them in any fashion that makes sense. For example, it will allow:

```
rhino           VAR     Word                        ' A 16-bit variable
eye             VAR     rhino.HIGHBYTE.LOWNIB.BIT1  ' A bit
```

All 2

All BS2 models can interpret twos complement negative numbers correctly in DEBUG and SEROUT instructions using formatters like SDEC (for signed decimal). In calculations, however, it assumes that all values are positive. This yields correct results with two's complement negative numbers for addition, subtraction, and multiplication, but not for division.

UNARY AND BINARY OPERATORS.

The standard operators we just discussed: +, - ,* and / all work on two values; as in 1 + 3 or 26 * 144. The values that operators process are referred to as arguments. So we say that the add, subtract, multiply and divide operators take two arguments.

Operators that take two arguments are called "binary" operators, and those that take only one argument are called "unary" operators. Please note that the term "binary operator" has nothing to do with binary numbers; it's just an inconvenient coincidence that the same word, meaning 'involving two things' is used in both cases.

The minus sign (-) is a bit of a hybrid. It can be used as a binary operator, as in 8-2 = 6, or it can be used as a unary operator to represent negative numbers, such as -4.

Unary operators take precedence over binary operators; the unary operation is always performed first. For example, on all BS2 models, SQR is the unary operator for square root. In the expression 10 - SQR 16, the BASIC Stamp first takes the square root of 16, then subtracts it from 10.

NOTES ABOUT THE 16-BIT WORKSPACE.

Most of the descriptions that follow say something like "computes (some function) of a 16-bit value." This does not mean that the operator does not work on smaller byte or nibble values, but rather that the computation is done in a 16-bit workspace. If the value is smaller than 16 bits, the BASIC Stamp pads it with leading 0s to make a 16-bit value. If the 16-bit result of a calculation is to be packed into a smaller variable, the higher-order bits are discarded (truncated).

Keep this in mind, especially when you are working with two's complement negative numbers, or moving values from a larger variable to a smaller one. For example, look at what happens when you move a two's complement negative number into a byte (rather than a word):

For example:

```
SYMBOL    value1   =  W0
SYMBOL    value2   =  W1

value1 = - 99
value2 = 100
value1 = value1 + value2              ' Add the numbers
DEBUG  value1                         ' Show the result (1)
```

-- or --

```
value1    VAR   Word
value2    VAR   Word

value1 = - 1575
value2 = 976
value1 = value1 + value2              ' Add the numbers
DEBUG  SDEC  ?  value1                ' Show the result (-599)
```

SUBTRACT: -

The Subtraction operator (-) subtracts variables and/or constants, returning a 16-bit result. It works exactly as you would expect with unsigned integers from 0 to 65535. If the result is negative, it will be correctly expressed as a signed 16-bit number. For example:

```
SYMBOL    value1   =  W0
SYMBOL    value2   =  W1

value1 = 199
value2 = 100
value1 = value1 - value2              ' Subtract the numbers
DEBUG  value1                         ' Show the result (99)
```

-- or --

```
value1    VAR   Word
value2    VAR   Word

value1 = 1000
value2 = 1999
value1 = value1 - value2              ' Subtract the numbers
DEBUG  SDEC  ?  value1                ' Show the result (-999)
```

MULTIPLY: *

The Multiply operator (*) multiplies variables and/or constants, returning the low 16 bits of the result. It works exactly as you would expect with unsigned integers from 0 to 65535. If the result of multiplication is larger than 65535, the excess bits will be lost. Multiplication of signed variables will be correct in both number and sign, provided that the result is in the range -32767 to +32767.

hexadecimal, you might think it was 41, in decimal… a totally different number. To help avoid this, use the IHEX formatter (the "I" stands for indicated). Changing the DEBUG line to read: DEBUG IHEX x would print "$41" on the screen. A similar formatter for binary also exists, IBIN, which prints a "%" before the number.

DISPLAYING SIGNED VS. UNSIGNED NUMBERS.

Signed numbers are preceded with a space ( ) or a minus sign (-) to indicate a positive or negative number, respectively. Normally, any number displayed by the BASIC Stamp is shown in its unsigned (positive) form without any indicator. The signed formatters allow you to display the number as a signed (rather than unsigned) value. **NOTE: Only Word-sized variables can be used for signed number display.** The code below demonstrates the difference in all three numbering schemes.

```
x       VAR     Word

x = -65
DEBUG  "Signed: ", SDEC  x, "  ", ISHEX  x, "  ", ISBIN  x, CR
DEBUG  "Unsigned: ", DEC  x, "  ", IHEX  x, "  ", IBIN  x
```

This code will generate the display shown below:

```
Signed:  -65    -$41    -%1000001
Unsigned:  65471    $FFBF    %1111111110111111
```

The signed form of the number –65 is shown in decimal, hexadecimal and then in binary on the top line. The unsigned form, in all three number systems, is shown on the bottom line. If the unsigned form looks strange to you, it's because negative numbers are stored in twos complement format within the BASIC Stamp.

AUTOMATIC NAMES IN THE DISPLAY.

Suppose that your program contained several DEBUG instructions showing the contents of different variables. You would want some way to tell them apart. One possible way is to do the following:

```
x       VAR     Byte
y       VAR     Byte

x = 100
y = 250
DEBUG "X = ", DEC x, CR                 ' Show decimal value of x
DEBUG "Y = ", DEC y, CR                 ' Show decimal value of y
```

TECHNICAL BACKGROUND

All 2 On all the BS2 models, DEBUG is actually a special case of the SEROUT instruction. It is set for inverted (RS-232-compatible) serial output through the programming connector (the SOUT pin) at 9600 baud, no parity, 8 data bits, and 1 stop bit. For example,

```
DEBUG "Hello"
```

is exactly like:

2
```
' {$STAMP BS2}

SEROUT 16, $4054, ["Hello"]
```

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

Another method to decrease program space is to reduce the number of DEBUG instructions by spreading DEBUG data across multiple lines. To do this, each line that wraps around must end with a comma as in the example below:

```
' {$PBASIC 2.5}

DEBUG "This is line 1", CR,
      "This is line 2"
```

The example above works identically to, but uses less program space than this version:

```
DEBUG "This is line 1", CR
DEBUG "This is line 2"
```

Note that spreading a DEBUG statement across multiple lines requires the declaration of PBASIC 2.5 syntax.

You may view DEBUG's output using a terminal program set to the above parameters, but you may have to modify either your development board or the serial cable to temporarily disconnect pin 3 of the BASIC Stamp (pin 4 of the DB-9 connector). See the SEROUT command for more detail.

A demo program for all BS2 models that uses DEBUG and DEBUGIN commands can be found at the end of the DEBUGIN section, next.

FOR…NEXT – BASIC Stamp Command Reference

on the BS1) will appear on the pin. The demo program shows how this works.

Demo Program (INPUT.bs1)


```
' INPUT.bs1
' This program demonstrates how the input/output direction of a pin is
' determined by the corresponding bit of DIRS. It also shows that the
' state of the pin itself (as reflected by the corresponding bit of PINS)
' is determined by the outside world when the pin is an input, and by the
' corresponding bit of OUTS when it's an output. To set up the demo,
' connect a 10k resistor from +5V to P7 on the BASIC Stamp. The resistor
' to +5V puts a high (1) on the pin when it's an input. The BASIC Stamp
' can override this state by writing a low (0) to bit 7 of OUTS and
' changing the pin to output.

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

Main:
  INPUT 7                               ' Make P7 an input
  DEBUG "State of P7: ", #PIN7, CR

  PIN7 = 0                              ' Write 0 to output latch
  DEBUG "After 0 written to OUT7: "
  DEBUG #PIN7, CR

  OUTPUT 7                              ' Make P7 an output
  DEBUG "After P7 changed to output: "
  DEBUG #PIN7
```

Demo Program (INPUT.bs2)

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

```
' INPUT.bs2
' This program demonstrates how the input/output direction of a pin is
' determined by the corresponding bit of DIRS. It also shows that the
' state of the pin itself (as reflected by the corresponding bit of INS)
' is determined by the outside world when the pin is an input, and by the
' corresponding bit of OUTS when it's an output. To set up the demo,
' connect a 10k resistor from +5V to P7 on the BASIC Stamp. The resistor
' to +5V puts a high (1) on the pin when it's an input. The BASIC Stamp
' can override this state by writing a low (0) to bit 7 of OUTS and
' changing the pin to output.

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

Main:
  INPUT 7                               ' Make P7 an input
  DEBUG "State of P7: ",
```

```
  LCDCMD Lcd, %00110000         ' send wakeup sequence to LCD
  PAUSE 5                       ' pause required by LCD specs
  LCDCMD Lcd, %00110000
  PAUSE 0                       ' pause required by LCD specs
  LCDCMD Lcd, %00110000
  PAUSE 0                       ' pause required by LCD specs
  LCDCMD Lcd, %00100000         ' set data bus to 4-bit mode
  LCDCMD Lcd, %00101000         ' set to 2-line mode with 5x8 font
  LCDCMD Lcd, %00001100         ' display on without cursor
  LCDCMD Lcd, %00000110         ' auto-increment cursor

  LCDOUT Lcd, LcdCGRam,         ' load custom character map
         [$00, $0A, $0A, $00, $11, $0E, $06, $00]

Main:
  DO
    LCDOUT Lcd, LcdCls, ["Hello my friend."]
    PAUSE 750
    LCDOUT Lcd, LcdLine2, ["How are you?"]
    PAUSE 1500
    LCDCMD Lcd, LcdCls
    LCDOUT Lcd, LcdLine1 + 1, ["I'm doing just"]
    LCDOUT Lcd, LcdLine2 + 4, ["fine!  ", 0]
    PAUSE 2000
  LOOP
  END
```

POLLIN

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

POLLIN ***Pin, State***

Function

Specify a polled-input pin and active state.

- ***Pin*** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This I/O pin will be set to input mode.
- ***State*** is a variable/constant/expression (0 – 1) that specifies whether to poll the I/O pin for a low (0) or a high (1) level.

Quick Facts

Table 5.73: POLLIN Quick Facts.

	BS2p, BS2pe, and BS2px
Available actions in response to reaching the desired *State*	1) Nothing, 2) Set polled-output pins to a specified state, 3) Run another program (in a specified program-slot), 4) Wait (pause program execution) until desired *State* is reached, 5) Any combination of 2, 3 and 4, above.
Special notes	• The polled-input pins are monitored (polled) in-between each command within the PBASIC code. • On the BS2p40, polled-input pins can be defined on both Main I/O and Auxiliary I/O pins. These are all active regardless of which group the program happens to be using at the time of a polling event.
Useful SPRAM locations	Locations 128 – 135 hold polled interrupt status. See Table 5.77 in the POLLMODE command section for more information.
Related commands	POLLMODE, POLLOUT, POLLRUN and POLLWAIT

Explanation

The POLLIN command is used to specify an input pin to monitor, or "poll", in-between instructions during the rest of the PBASIC program. The BASIC Stamp will then perform some activity (in-between instructions) when the specified *State* is detected. The activity performed depends on the POLLMODE, POLLOUT and POLLRUN commands.

The "polling" commands allow the BASIC Stamp to respond to certain I/O pin events at a faster rate than what is normally possible through manual PBASIC programming. The term "poll" comes from the fact that the BASIC Stamp module's interpreter periodically checks the state of the designated polled-input pins. It "polls" these pins after the end of each PBASIC command and before it reads the next PBASIC command from the

At their heart, synchronous-serial devices are essentially shift-registers; trains of flip-flops that pass data bits along in a bucket brigade fashion to a single data output pin. Another bit is output each time the appropriate edge (rising or falling, depending on the device) appears on the clock line.

SHIFTIN OPERATION.

The SHIFTIN instruction first causes the clock pin to output low and the data pin to switch to input mode. Then, SHIFTIN either reads the data pin and generates a clock pulse (PRE mode) or generates a clock pulse then reads the data pin (POST mode). SHIFTIN continues to generate clock pulses and read the data pin for as many data bits as are required.

Making SHIFTIN work with a particular device is a matter of matching the mode and number of bits to that device's protocol. Most manufacturers use a timing diagram to illustrate the relationship of clock and data. Items to look for include: 1) which bit of the data arrives first; most significant bit (MSB) or least significant bit (LSB) and 2) is the first data bit ready before the first clock pulse (PRE) or after the first clock pulse (POST). Table 5.115 shows the values and symbols available for *Mode,* and Figure 5.42 shows SHIFTIN's timing.

Symbol	Value	Meaning
MSBPRE	0	Data is msb-first; sample bits before clock pulse
LSBPRE	1	Data is lsb-first; sample bits before clock pulse
MSBPOST	2	Data is msb-first; sample bits after clock pulse
LSBPOST	3	Data is lsb-first; sample bits after clock pulse

(Msb is most-significant bit; the highest or leftmost bit of a nibble, byte, or word. Lsb is the least-significant bit; the lowest or rightmost bit of a nibble, byte, or word.)

Table 5.115: SHIFTIN *Mode* Values and Symbols.

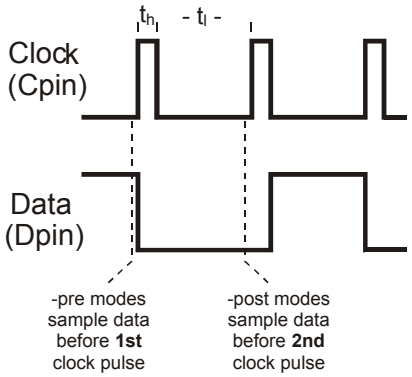


Figure 5.42: SHIFTIN Timing Diagram. Refer to the SHIFTIN Quick Facts table for timing information on t_h (t high) and t_l (t low).

SOUND

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

SOUND ***Pin*****, (** ***Note*****,** ***Duration*** **{ ,** ***Note*****,** ***Duration…*** **})**

(See FREQOUT)

Function

Generate square-wave tones for a specified period.

- ***Pin*** is a variable/constant (0 – 7) that specifies the I/O pin to use. This pin will be set to output mode.
- ***Note*** is a variable/constant (0 – 255) specifying the type and frequency of the tone. 1 – 127 are ascending tones and 128 – 255 are ascending white noises ranging from buzzing (128) to hissing (255).
- ***Duration*** is a variable/constant (1 - 255) specifying the amount of time to generate the tone(s). The unit of time for *Duration* is 12 ms.

Quick Facts

Table 5.119: SOUND Quick Facts.

	BS1
Units in *Duration*	12 ms
Available Sounds	256
Frequency Range	94.8 Hz to 10,550 Hz

Explanation

SOUND generates one of 256 square-wave frequencies on an I/O pin. The output pin should be connected as shown in Figure 5.46.

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

Note = 127 – (((1/Frequency)-0.000095)/0.000083)

--and--

Frequency = (1/(0.000095 + ((127–Note)*0.000083))

In the above equations, Frequency is in Hertz (Hz).

Demo Program (STORE0.bsp)

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

```
' STORE0.bsp
' This program demonstrates the STORE command and how it affects the READ
' and WRITE commands.  This program "STORE0.BSP" is intended to be down-
' loaded into program slot 0.  It is meant to work with STORE1.BSP and
' STORE2.BSP.  Each program is very similar (they display the current
' Program Slot and READ/WRITE Slot numbers and the values contained in the
' first five EEPROM locations.  Each program slot will have different data
' due to different DATA commands in each of the programs downloaded.

' {$STAMP BS2p, STORE1.BSP, STORE2.BSP}
' {$PBASIC 2.5}

#IF ($STAMP < BS2P) #THEN
  #ERROR "This program requires BS2p, BS2pe, or BS2px."
#ENDIF

idx             VAR     Word            ' index
value           VAR     Byte

LocalData       DATA    @0, 1, 2, 3, 4, 5


Main:
  GOSUB Show_Slot_Info                  ' show slot info/data
  PAUSE 2000
  STORE 1                               ' point READ/WRITE to Slot 1
  GOSUB Show_Slot_Info
  PAUSE 2000
  RUN 1                                 ' run program in Slot 1
  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
```

— F —

— G —

— H —

— I —