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/pbasic48w-p24

(such as two BS2s) on two ports and you have two different PBASIC programs to download (one to each BS2). Without this directive, developing and downloading in this case would be a tedious task of always answering the "which BASIC Stamp?" prompt.

The $PORT directive can be automatically inserted or modified by selecting the appropriate port from the Directive → Port menu. The COM ports listed in the Directive → Port menu are automatically updated any time a change is made to the exiting computer hardware or to the available ports list. See the Setting Preferences section which begins on page 55 for more information.

Special Functions

THE IDENTIFICATION FUNCTION.

The Identify function will identify which BASIC Stamp model, if any, is detected on any available communications port. This information is displayed in the Identification window (Figure 3.10), which can greatly aid in troubleshooting your connection to your BASIC Stamp module. Activate this function by selecting Run → Identify, by pressing Ctrl-I, or pressing F6.

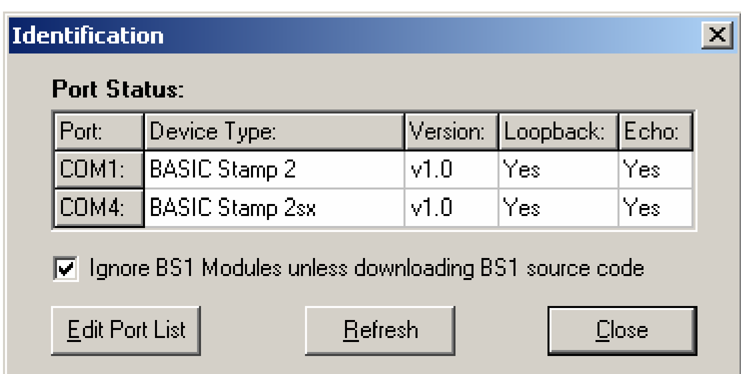

Figure 3.10: The Identification Window.

The Port column shows the available ports (those that the BASIC Stamp Editor is trying to access). You can modify the available Port List by clicking on the Edit Port List button. Modifying this list only affects which ports the BASIC Stamp Editor tries to use; it does not affect which serial ports are installed on your computer. It is recommended that you delete all known modem ports and any problematic ports from this list.

CUSTOMIZED SYNTAX HIGHLIGHTING.

To create a custom scheme, select a default scheme you wish to modify, and click on the Copy Scheme button. Then, select (highlight) an element within the Syntax Element list, and apply new Text Attributes with the checkboxes and drop-down menus to the right. As you try various text attributes and color combinations, the Show Preview Example checkbox lets you audition your custom scheme without closing the Preferences window.

The BASIC Stamp Editor supports one custom scheme at a time. It can be modified indefinitely, but it cannot be copied. If you again copy a default scheme, you will be asked to confirm that you wish to overwrite your current custom scheme.

Under this tab, you will also find checkboxes that allow you to show or hide bookmarks, line numbers, the overwrite cursor, and the toolbar.

EDITOR OPERATION PREFERENCES.

Under the Editor Operation tab (Figure 3.19), you may set preferences for automatic indentation and tab behavior.

AUTO INDENTING / UNINDENTING.

The Auto Indent on Enter option makes it easy to indent nested loops to make code easier to read. The Auto Unindent option enables quick reversal of an indented line by simply using the backspace key, provided that the cursor is to the left of the first character on the line.

TAB CHARACTER.

The editor lets you choose whether a tab character or spaces are inserted into source code whenever you press the Tab key. The default setting, insert space characters upon Tab key presses, is recommended because it enforces the intended formatting regardless of what editor you use to view the code later.

BASIC Stamp Architecture Introduction This chapter provides detail on the architecture (RAM usage) and math functions of the BS1, BS2, BS2e, BS2sx, BS2p, BS2pe, and BS2px.

The following icons will appear to indicate where there are differences among the various BASIC Stamp models:

One or more of these icons indicates the item applies only to the BS1, BS2, BS2e, BS2sx, BS2p, BS2pe, or BS2px respectively.

If an item applies to the all of the models in the BS2 family, this icon is used.

MEMORY ORGANIZATION

The BASIC Stamp has two kinds of memory; RAM (for variables used by your program) and EEPROM (for storing the program itself). EEPROM may also be used to store long-term data in much the same way that desktop computers use a hard drive to hold both programs and files.

An important distinction between RAM and EEPROM is this:

- RAM loses its contents when the BASIC Stamp loses power; when power returns, all RAM locations are cleared to 0s.
- EEPROM retains the contents of memory, with or without power, until it is overwritten (such as during the program-downloading process or with a WRITE instruction.)

RAM ORGANIZATION (BS1)

The BS1 has 16 bytes (8 words) of RAM space arranged as shown in Table 4.1 The first word, called PORT, is used for I/O pin control. It consists of two bytes, PINS and DIRS. The bits within PINS correspond to each of the eight I/O pins on the BS1. Reading PINS effectively reads the I/O pins directly, returning an 8-bit set of 1's and 0's corresponding to the high and low state of the respective I/O pin at that moment. Writing to PINS will store a high or low value on the respective I/O pins (though only on pins that are set to outputs).

THE INPUT/OUTPUT VARIABLES.

The second byte of PORT, DIRS, controls the direction of the I/O pins. Each bit within DIRS corresponds to an I/O pin's direction. A high bit (1)

All 2

```
result          VAR     Word

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

ENCODER: NCD

All 2

The Encoder operator (NCD) is a "priority" encoder of a 16-bit value. NCD takes a 16-bit value, finds the highest bit containing a 1 and returns the bit position plus one (1 through 16). If the input value is 0, NCD returns 0. NCD is a fast way to get an answer to the question "what is the largest power of two that this value is greater than or equal to?" The answer NCD returns will be that power, plus one. Example:

```
result          VAR     Word

result = %1101                          ' Highest bit set is bit 3
DEBUG ? NCD result                      ' Show the NCD of result (4)
```

SINE: SIN

All 2

The Sine operator (SIN) returns the two's complement, 16-bit sine of an angle specified as an 8-bit binary radian (0 to 255) angle.

To understand the SIN operator more completely, let's look at a typical sine function. By definition: given a circle with a radius of 1 unit (known as a unit circle), the sine is the y-coordinate distance from the center of the circle to its edge at a given angle. Angles are measured relative to the 3-o'clock position on the circle, increasing as you go around the circle counterclockwise.

At the origin point (0 degrees) the sine is 0, because that point has the same y (vertical) coordinate as the circle center. At 45 degrees the sine is 0.707. At 90 degrees, sine is 1. At 180 degrees, sine is 0 again. At 270 degrees, sine is -1.

The BASIC Stamp SIN operator breaks the circle into 0 to 255 units instead of 0 to 359 degrees. Some textbooks call this unit a "binary radian" or "brad." Each brad is equivalent to 1.406 degrees. And instead of a unit circle, which results in fractional sine values between 0 and 1, BASIC Stamp SIN is based on a 127-unit circle. Results are given in two's complement form in order to accommodate negative values. So, at the origin, SIN is 0. At 45 degrees (32 brads), sine is 90. At 90 degrees (64 brads), sine is 127. At 180 degrees (128 brads), sine is 0. At 270 degrees (192 brads), sine is -127.

Table 5.28: Layout of SPRAM Registers.

NOTE: Scratch Pad RAM can only be accessed with the GET and PUT commands. Scratch Pad RAM cannot have variable names assigned to it.

Location	BS2e and BS2sx	BS2p, BS2pe, and BS2px
0...62	General Purpose RAM	General Purpose RAM
63	Bits 0-3: Active program slot number.	General Purpose RAM
64..126	n/a	General Purpose RAM
127	n/a	Bits 0-3, Active program slot #. Bits 4-7, program slot for READ and WRITE operations.
128	n/a	Polled input trigger status of Main I/O pins 0-7 (0 = not triggered, 1 = triggered).
129	n/a	Polled input trigger status of Main I/O pins 8-15 (0 = not triggered, 1 = triggered).
130	n/a	Polled input trigger status of Auxiliary I/O pins 0-7 (0 = not triggered, 1 = triggered).
131	n/a	Polled input trigger status of Auxiliary I/O pins 8-15 (0 = not triggered, 1 = triggered).
132	n/a	Bits 0-3: Polled-interrupt mode, set by POLLMODE
133	n/a	Bits 0-2: Polled-interrupt "run" slot, set by POLLRUN.
134	n/a	Bit 0: Active I/O group; 0 = Main I/O, 1 = Auxiliary I/O.
135	n/a	Bit 0: Polled-output status (set by POLLMODE); 0 = disabled, 1= enabled. Bit 1: Polled-input status; 0 = none defined, 1 = at least one defined. Bit 2: Polled-run status (set by POLLMODE); 0 = disabled, 1 = enabled. Bit 3: Polled-output latch status; 0 = real-time mode, 1 = latch mode. Bit 4: Polled-input state; 0 = no trigger, 1 = triggered. Bit 5: Polled-output latch state; 0 = nothing latched, 1 = signal latched. Bit 6: Poll-wait state; 0 = No Event, 1 = Event Occurred. (Cleared by POLLMODE only). Bit 7: Polling status; 0 = not active, 1 = active.

2e 2sx 2p 2pe 2px

Demo Program (GET_PUT1.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_PUT1.bsx
' This example demonstrates the use of the GET and PUT commands.  First,
' slot location is read using GET to display the currently running program
' number.  Then a set of values are written (PUT) into locations 0 TO 9.
' Afterwards, program number 1 is RUN.  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.

' {$STAMP BS2sx, GET_PUT2.BSX}
' {$PBASIC 2.5}
```

BASIC Stamp encounters a RETURN without a previous GOSUB, the entire program starts over from the beginning. Take care to avoid these phenomena.

1

Demo Program (GOSUB.bs1)

```
' GOSUB.bs1
' This program is a guessing game that generates a random number in a
' subroutine called Pick_A_Number. It is written to stop after ten
' guesses. To see a common bug associated with GOSUB, delete or comment
' out the line beginning with END after the FOR-NEXT loop. This means
' that after the loop is finished, the program will wander into the
' Pick_A_Number subroutine. When the RETURN at the end executes, the
' program will go back to the beginning of the program. This will cause
' the program to execute endlessly. Make sure that your programs can't
' accidentally execute subroutines!

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

SYMBOL  rounds          = B2            ' number of reps
SYMBOL  numGen          = W0            ' random number holder
SYMBOL  myNum           = B3            ' random number, 1-10


Setup:
  numGen = 11500                        ' initialize random "seed"

Main:
  FOR rounds = 1 TO 10
    DEBUG CLS, "Pick a number from 1 to 10", CR
    GOSUB Pick_A_Number
    PAUSE 2000                          ' dramatic pause
    DEBUG "My number was: ", #myNum     ' show the number
    PAUSE 1000                          ' another pause.
  NEXT
  DEBUG CLS, "Done"
  END                                   ' end program

' Random-number subroutine. A subroutine is just a piece of code with
' the RETURN instruction at the end. Always make sure your program enters
' subroutines with a GOSUB. If you don't, the RETURN won't have the
' correct address, and your program will have a bug!

Pick_A_Number:
  RANDOM numGen                         ' stir up the bits of NumGen.
  DEBUG numGen, CR
  myNum = numGen / 6550 MIN 1           ' scale to fit 1-10 range.
  RETURN                                ' go back to 1st instruction
                                        ' after GOSUB that got us here
```

Demo Program (GOSUB.bs2)

All 2

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

```
' GOSUB.bs2
' This program is a guessing game that generates a random number in a
' subroutine called Pick_A_Number. It is written to stop after ten
' guesses. To see a common bug associated with GOSUB, delete or comment
' out the line beginning with END after the FOR-NEXT loop. This means
' that after the loop is finished, the program will wander into the
' Pick_A_Number subroutine. When the RETURN at the end executes, the
' program will go back to the beginning of the program. This will cause
' the program to execute endlessly. Make sure that your programs can't
' accidentally execute subroutines!

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

rounds          VAR     Byte            ' number of reps
numGen          VAR     Word            ' random number holder
myNum           VAR     Byte            ' random number, 1-10


Setup:
  numGen = 11500                        ' initialize random "seed"

Main:
  FOR rounds = 1 TO 10
    DEBUG CLS, "Pick a number from 1 to 10", CR
    GOSUB Pick_A_Number
    PAUSE 2000                          ' dramatic pause
    DEBUG "My number was: ", DEC myNum  ' show the number
    PAUSE 1000                          ' another pause.
  NEXT
  DEBUG CLS, "Done"
  END                                   ' end program

' Random-number subroutine. A subroutine is just a piece of code with
' the RETURN instruction at the end. Always make sure your program enters
' subroutines with a GOSUB. If you don't, the RETURN won't have the
' correct address, and your program will have a bug!

Pick_A_Number:
  RANDOM numGen                         ' stir up the bits of NumGen.
  DEBUG DEC ? numGen
  myNum = numGen / 6550 MIN 1           ' scale to fit 1-10 range.
  RETURN                                ' go back to 1st instruction
                                        ' after GOSUB that got us here
```

Quick Facts

Table 5.35: I2COUT Quick Facts.

	BS2p, BS2pe, and BS2px	
Values for *Pin*	*Pin* = 0	*Pin* = 8
I/O Pin Arrangement	0: Serial Data (SDA) pin 1: Serial Clock (SCL) pin	8: Serial Data (SDA) pin 9: Serial Clock (SCL) pin
Transmission Rate	Approximately 81 kbits/sec on a BS2p, 45 kbits/sec on a BS2pe, and 83 kbits/sec on a BS2px (not including overhead).	
Special Notes	The SDA and SCL pins must have 1 kΩ - 4.7 kΩ pull-up resistors. The I2CIN command does not allow for multiple masters. The BASIC Stamp cannot operate as an I^{2}C slave device.	
Related Command	I2CIN	

Explanation

The I^{2}C protocol is a form of synchronous serial communication developed by Phillips Semiconductors. It only requires two I/O pins and both pins can be shared between multiple I^{2}C devices. The I2COUT command allows the BASIC Stamp to send data to an I^{2}C device.

A SIMPLE I2COUT EXAMPLE.

The following is an example of the I2COUT command:

```
I2COUT 0, $A0, 5, [100]
```

This code will transmit a "write" command to an I^{2}C device (connected to I/O pins 0 and 1), followed by an address of 5 and finally will transmit the number 100.

The above example will write a byte of data to location 5 of a 24LC16B EEPROM from Microchip. Figure 5.11 shows the proper wiring for this example to work. The *SlaveID* argument ($A0) is both the ID of the chip and the command to write to the chip; the 0 means write. The *Address* argument (5) is the EEPROM location to write to.

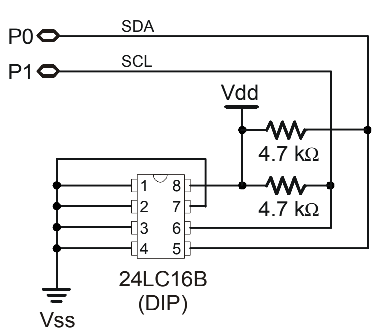

Figure 5.11: Example Circuit for the I2COUT command and a 24LC16B EEPROM.

Note: The 4.7 kΩ resistors are required for the I2COUT command to function properly.

Stamp that all commands following it should affect the auxiliary I/O pins (*Port = 1*). The following LOW command will set I/O pin 0 of the auxiliary I/O pins (physical pin 21) low.

Note that the main I/O and auxiliary I/O pins are independent of each other; the states of the main I/O pins remain unchanged while the program affects the auxiliary I/O pins, and vice versa.

MAIN I/O AND AUXILIARY I/O PINS ARE INDEPENDENT AND UNAFFECTED BY CHANGES IN THE OPPOSITE GROUP.

Other commands that affect I/O group access are AUXIO and MAINIO.

Demo Program (AUX_MAIN_TERM.bsp)

2p

NOTE: This example program will tokenize with the 24-pin BS2p, BS2pe, and BS2px but its effects can only be seen on the BS2p40. This program uses conditional compilation techniques; see Chapter 3 for more information.

```
' AUX_MAIN_TERM.bsp
' This program demonstrates the use of the AUXIO, MAINIO and IOTERM
' commands to affect I/O pins in the auxiliary and main I/O groups.

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

#SELECT $STAMP
  #CASE BS2, BS2E, BS2SX
    #ERROR "Program requires BS2p40"
  #CASE BS2P, BS2PE, BS2PX
    DEBUG "Note: This program designed for the BS2p40.", CR
#ENDSELECT

port            VAR     Bit


Main:
  DO
    MAINIO                              ' Switch to main I/O pins
    TOGGLE 0                            ' Toggle state of I/O pin P0
    PWM 1, 100, 40                      ' Generate PWM on I/O pin P1

    AUXIO                               ' Switch to auxiliary I/O pins
    TOGGLE 0                            ' Toggle state of I/O pin X0
    PULSOUT 1, 1000                     ' Generate a pulse on I/O pin X1
    PWM 2, 100, 40                      ' Generate PWM on I/O pin X2

    IOTERM port                         ' Switch to main or aux I/Os
                                        ' -- depending on port
    TOGGLE 3                            ' Toggle state of I/O pin 3
                                        ' -- on main and aux, alternately
    port = ~port                        ' Invert port
    PAUSE 1000                          ' 1 second delay
  LOOP
  END
```

printed on the display (with the LCDOUT command) will appear at the current cursor's location. Here's another example:

```
LCDCMD 0, 128 + 64
```

The above command will move the cursor to the first character position on the second line (on a 2 x 16 display). 128 is the Move To Display Address command and 64 is the location number. See the "Character Positioning" section, below, for more information.

	Command (in decimal)	Description
Do Nothing	0	Don't perform any special operation.
Clear Display	1	Clear the display and move cursor to home position.
Home Display	2	Move cursor and display to home position.
Inc Cursor	6	Set cursor direction to right, without a display shift.
Display Off	8	Turn off display (display data is retained).
Display On	12	Turn on display without cursor (display is restored).
Blinking Cursor	13	Turn on display with blinking cursor.
Underline Cursor	14	Turn on display with underline cursor.
Cursor Left	16	Move cursor left one character.
Cursor Right	20	Move cursor right one character.
Scroll Left	24	Scroll display left one character.
Scroll Right	28	Scroll display right one character.
Move To CGRAM Address	64 + address	Move pointer to character RAM location.
Move To DDRAM Address	128 + address	Move cursor to Display Data RAM location.

Table 5.44: Common LCD Commands. These are supported by LCDs with the Hitachi 44780 controller.

A NOTE ABOUT ADVANCED LCD COMMANDS.

While most users will only need the commands shown in Table 5.44 above, Table 5.45 below details all of the instructions supported by the LCD (for advanced users). Many instructions are multipurpose, depending on the state of special bits. Clever manipulation of the instruction bits will allow for powerful control of the LCD.

CHARACTER POSITIONING: MOVING THE CURSOR.

The last command shown above (Move To DDRAM Address) is used to move the cursor to a specific position on the LCD. The LCD's DDRAM (Display Data RAM) is a fixed size with a unique position number for each character cell. The viewable portion of the DDRAM depends on the LCD's logical view position (which can be altered with the Scroll Display command). The default view position is called the Home position; it means that the display's upper left character corresponds to DDRAM

Table 5.48: LCDIN Conversion Formatters

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

1 All numeric conversions will continue to accept new data until receiving either the specified number of digits (ex: three digits for DEC3) or a non-numeric character.
2 To be recognized as part of a number, the minus sign (-) must immediately precede a numeric character. The minus sign character occurring in non-numeric text is ignored and any character (including a space) between a minus and a number causes the minus to be ignored.
3 The hexadecimal formatters are not case-sensitive; "a" through "f" means the same as "A" through "F".
4 Indicated hexadecimal and binary formatters ignore all characters, even valid numerics, until they receive the appropriate prefix ($ for hexadecimal, % for binary). The indicated formatters can differentiate between text and hexadecimal (ex: ABC would be interpreted by HEX as a number but IHEX would ignore it unless expressed as $ABC). Likewise, the binary version can distinguish the decimal number 10 from the binary number %10. A prefix occurring in non-numeric text is ignored, and any character (including a space) between a prefix and a number causes the prefix to be ignored. Indicated, signed formatters require that the minus sign come before the prefix, as in -$1B45.
5 The HEX modifier can be used for Decimal to BCD Conversion. See "Hex to BCD Conversion" on page 97.

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

LOOKDOWN

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

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

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

Function

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

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

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

All 2

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

Quick Facts

Table 5.52: LOOKDOWN Quick Facts.

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

Explanation

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

```
          SDEC tempC, " C ", CR,
          SDEC tempF, " F "
     PAUSE 1000
  LOOP
  END

Get_Temperature:
  OWOUT DQ, 1, [SkipROM, CvrtTmp]        ' send convert temperature command
  DO                                     ' wait on conversion
    PAUSE 25                             ' small loop pad
    OWIN DQ, 4, [tempIn]                 ' check status (bit transfer)
  LOOP UNTIL (tempIn)                    ' 1 when complete
  OWOUT DQ, 1, [SkipROM, RdSP]           ' read DS1822 scratch pad
  OWIN  DQ, 2, [tLo, tHi]                ' get raw temp data
  tSign = sign                           ' save sign bit
  tempC = tempIn >> 4                    ' round to whole degrees
  tempC.BYTE1 = $FF * tSign              ' correct twos complement bits
  tempF = (ABS tempC) * 9 / 5            ' start F conversion
  IF (tSign) THEN                        ' finish F conversion
    tempF = 32 - tempF                   ' C was negative
  ELSE
    tempF = tempF + 32                   ' C was positive
  ENDIF
  RETURN
```

```
' Download this program to Slot 0

DEBUG "Hello "
RUN 1

' Download this program to Slot 1

DEBUG "World!"
PAUSE 1000
RUN 0
```

The above two programs (assuming they have been downloaded into program slots 0 and 1, respectively) will display "Hello World!" on the screen. Program 0 is the first to run and it displays "Hello ", then issues a RUN 1 command. The BASIC Stamp then starts execution of program 1, from its first line of code, which causes "World!" to be displayed. Program 1 then pauses for 1 second and the runs program 0 again.

WHAT HAPPENS TO I/O PINS AND RAM WHEN USING RUN?

The I/O pins retain their current state (directions and output latches) and all RAM and SPRAM locations retain their current data during a transition between programs with the RUN command. If sharing data between programs within RAM, make sure to keep similar variable declarations (defined in the same order) in all programs so that the variables align themselves on the proper word, byte, nibble and bit boundaries across programs. The following programs illustrate what happens with mismatched variable declarations:

```
' Download this program to Slot 0
cats    VAR     Byte
dogs    VAR     Byte

Setup:
  cats = 3
  dogs = 1
  DEBUG ? cats
  DEBUG ? dogs
  RUN 1

' Download this program to Slot 1
cats    VAR     Byte
dogs    VAR     Byte
fleas   VAR     Word

Main:
  DEBUG ? cats
  DEBUG ? dogs
  DEBUG ? fleas
  END
```

All 2

```
' {$PBASIC 2.5}

result          VAR     Word

Main:
  DO
    SERIN 1, 24660, Bad_Data, 10000, No_Data, [DEC result]
    DEBUG CLS, ? result
  LOOP

Bad_Data:
  DEBUG CLS, "Parity error"
  GOTO Main

No_Data:
  DEBUG CLS, "Timeout error"
  GOTO Main
```

CONTROLLING DATA FLOW.

When you design an application that requires serial communication between BASIC Stamp modules, you have to work within these limitations:

- When the BASIC Stamp is sending or receiving data, it can't execute other instructions.
- When the BASIC Stamp is executing other instructions, it can't send or receive data. *The BASIC Stamp does not have a serial buffer* as there is in PCs. At most serial rates, the BASIC Stamp cannot receive data via SERIN, process it, and execute another SERIN in time to catch the next chunk of data, unless there are significant pauses between data transmissions.

These limitations can sometimes be addressed by using flow control; the *Fpin* option for SERIN and SEROUT (at baud rates of up to the limitation shown in Table 5.94). Through *Fpin*, SERIN can tell a BASIC Stamp sender when it is ready to receive data. (For that matter, *Fpin* flow control follows the rules of other serial handshaking schemes, but most computers other than the BASIC Stamp cannot start and stop serial transmission on a byte-by-byte basis. That's why this discussion is limited to communication between BASIC Stamp modules.)

Here's an example using flow control on the BS2 (data through I/O pin 1, flow control through I/O pin 0, 9600 baud, N8, noninverted):

All 2

```
serData         VAR     Byte

SERIN 1\0, 84, [serData]
```

All 2

```
SEROUT 1, 16780, ["Hello", CR]
SEROUT 1, 16780, ["Num = ", DEC 100]
```

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

The above code will display "HELLO" on one line and "Num = 100" on the next line. Notice that you can combine data to output in one SEROUT command, separated by commas. In the example above, we could have written it as one line of code, with "HELLO", CR, "Num = ", DEC 100 in the *OutputData* list.

1

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

USING SEROUT'S *PACE* ARGUMENT TO INSERT DELAYS BETWEEN TRANSMITTED BYTES.

All 2

NOTE: The rest of the code examples for this section are written for the BS2, using the BS2's *Baudmode* and *Timeout* values. Be sure to adjust the value for your BASIC Stamp model.

The SEROUT command can also be configured to pause between transmitted bytes. This is the purpose of the optional *Pace* argument. For example (9600 baud N8, inverted):

```
SEROUT 1, 16780, 1000, ["Slowly..."]
```

Here, the BASIC Stamp transmits "Slowly..." with a 1 second delay between each character. See Table 5.104 for units of the *Pace* argument. One good reason to use the *Pace* feature is to support devices that require more than one stop bit. Normally, the BASIC Stamp sends data as fast as it can (with a minimum of 1 stop bit between bytes). Since a stop bit is really just a resting state in the line (no data transmitted), using the *Pace* option will effectively add multiple stop bits. Since the requirement for 2 or more stop bits (on some devices) is really just a "minimum" requirement, the receiving side should receive this data correctly.

USING ASCII CODES.

Keep in mind that when we type something like "XYZ" into the SEROUT command, the BASIC Stamp actually uses the ASCII codes for each of those characters for its tasks. We could also typed: 88, 89, 90 in place of "XYZ" and the program would run the same way since 88 is the ASCII code for the "X" character, 89 is the ASCII code for the "Y" character, and so on.

ADDITIONAL CONVERSION FORMATTERS.

The decimal formatter is only one of a whole family of conversion formatters available with SERIN on all BS2 models. See Table 5.110 for a list of available conversion formatters. All of the conversion formatters work similar to the decimal formatter. The formatters translate the value into separate bytes of data until the entire number is translated or until the