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/pbasic2-p

Contents

automatically opened by the editor, however, if manually opened, these fields will be enabled to allow manual configuration. The signal status LEDs turn bright green when activity on the indicated port line is detected. The signal checkboxes (DTR and RTS) can be selected to set or clear the respective output line on the port.

The Echo Off checkbox (bottom of window) causes the Receive pane to throw away the characters that arrive in the port's receive buffer immediately after transmitting characters from the transmit buffer. This produces a cleaner Receive pane display for interactive programs such as the example above. Keep in mind, however, that this feature does not verify that the character it throws away is actually a match to a character that was just transmitted (because data collisions on the port can cause echoed characters to be garbled). You should only use the Echo Off feature in situations where it is required, as it may result in a strange display in certain applications.

KEYBOARD SHORTCUTS FOR CODING FUNCTIONS.

There are keyboard shortcuts for several coding functions, some of which are unique to the BASIC Stamp Editor.

Table 3.6: Coding Function Keyboard Shortcuts.

Coding Functions	
Shortcut Key(s)	**Function**
Ctrl+J	Show code templates.
F6 or Ctrl+I	Identify BASIC Stamp firmware.
F7 or Ctrl+T	Perform a syntax check on the code and display any error messages.
F8 or Ctrl+M	Open Memory Map window.
F9 or Ctrl+R	Tokenize code, download to the BASIC Stamp and open Debug window if necessary.
F11 or Ctrl+D	Open a new Debug window.
F12	Switch to next window (Editor, Debug #1, Debug #2, Debug #3 or Debug #4)
Ctrl+1, Ctrl+2, Ctrl+3, Ctrl+4	Switch to Debug Terminal #1, Debug Terminal #2, etc. if that Terminal window is open.
Ctrl+`	Switch to Editor window.
ESC	Close current window.

HELP FILES.

The BASIC Stamp Editor includes searchable, indexed help files. Access Help by selecting Help → Contents or Help → Index. Context sensitive help (highlighting a word in the editor and pressing F1 key) is also supported. The help file can remain open in a separate window while using the BASIC Stamp Editor; simply press Alt+Tab to toggle back and forth between the editor and the Help window.

#SELECT...#CASE SYNTAX.

#SELECT Expression
  #CASE ***Condition(s)***
    Statement(s)
 { **#CASE** ***Condition(s)***
    Statement(s)
  #CASE #ELSE
    Statement(s) }
#ENDSELECT

#SELECT…#CASE is a conditional compile structure similar to the run-time SELECT…CASE command except that, at compile time, #SELECT…#CASE evaluates *Expression* and then conditionally compiles a block of code based on comparison to *Condition(s)*. If no C*onditions* are found to be True and a #CASE #ELSE block is included, the *Statement(s)* in the #CASE #ELSE block will be compiled.

- ***Expression*** is a statement that can be evaluated as True or False during compile-time.
- ***Condition*** is a statement, that when compared to *Expression*, can be evaluated as True or False. Multiple conditions within the same CASE can be separated by commas (,).
- ***Statement*** is any valid PBASIC instruction.

Example:

```
' {$PBASIC 2.5}

#SELECT $STAMP
  #CASE BS2, BS2e, BS2sx
    GOSUB LCD_Write
  #CASE #ELSE
    LCDOUT LCDpin, cmd, [char]
#ENDSELECT
```

This example checks the $STAMP directive at compile-time and either compiles

```
GOSUB LCD_Write
```

- or –

`LCDOUT LCDpin, cmd, [char]` into the program.

All 2
```
' {$PBASIC 2.5}

signal    PIN   2       ' pin-type symbol representing I/O 2

OUTPUT  signal          ' set signal pin to output
signal = 1              ' set signal high
```

The OUTPUT command treats *signal* as a constant equal to 2 and the signal = 1 statement treats *signal* as a variable equal to the output variable for the defined pin (OUT2 in this case).

You might be wondering why "signal = 0" in the IF…THEN statement of our first example treats *signal* as the input variable *IN1* and yet "signal = 1" in our last example treats *signal* as the output variable *OUT2*. The distinction is that the first example is a comparison and the second example is an assignment. Comparisons need to "read" expressions and then evaluate the comparison while assignments need to read expressions and then "write" the results. Since *signal* is to the left of the equal sign (=) in our assignment statement, it must be a variable we can write to, thus it must be treated as OUT2, in this case.

What happens if our pin-type symbol is to the right of the equal sign in an assignment statement? Example:

All 2
```
' {$PBASIC 2.5}

signal1   PIN   1       ' pin-type symbol representing I/O 1
signal2   PIN   2       ' pin-type symbol representing I/O 2

INPUT  signal1          ' set signal1 pin to input
OUTPUT signal2          ' set signal2 pin to output
signal2 = signal1       ' set signal2 pin to signal1 pin's state
```

In this case *signal2* is treated as OUT2 and *signal1* is treated as IN1; left side must be written to and right side must be read from.

If a pin-type symbol is used in a command, but not in the *Pin* argument of that command, it will be treated as an input variable (i.e.: INx). NOTE: It is very rare that you'll need to use a pin-type symbol in this way.

The following is a summary of behaviors and uses of pin-type symbols.

DEBUG – BASIC Stamp Command Reference

Demo Program (DEBUGIN.bs2)

```
' DEBUGIN.bs2
' This program demonstrates the ability to accept user input from the
' Debug Terminal, and to accept numeric entry in any valid format.

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

myNum   VAR     Word


Main:
  DO
    DEBUG CLS, "Enter any number: "     ' prompt user
    DEBUGIN SNUM myNum                  ' retrieve number in any format

    DEBUG CLS,                          ' display number in all formats
          SDEC ? myNum,
          SHEX ? myNum,
          SBIN ? myNum
    PAUSE 3000
  LOOP                                  ' do it again
  END
```

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

IF…THEN – BASIC Stamp Command Reference

Quick Facts

	BS1	All BS2 Models
Comparison Operators	=, <>, >, <, >=, <=	=, <>, >, <, >=, <=
Conditional Logic Operators	AND, OR	NOT, AND, OR, XOR
Format of *Condition*	*Variable Comparison Value*; where *Value* is a variable or constant	*Value1 Comparison Value2*; where *Value1* and *Value2* can by any of variable, constant or expression
Parentheses	Not Allowed	Allowed
Max nested IF…THENs	n/a	16
Max ELSEIFs per IF	n/a	16
Max ELSEs per IF	n/a	1
Related Command	None	SELECT…CASE

Table 5.38: IF...THEN Quick Facts.

Explanation

IF...THEN is PBASIC's decision maker that allows one block of code or another to run based on the value (True or False) of a condition. The condition that IF...THEN tests is written as a mixture of comparison and logic operators. The available comparison operators are:

Comparison Operator Symbol	Definition
=	Equal
<>	Not Equal
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To

Table 5.39: IF...THEN Comparison Operators.

Comparisons are always written in the form: Value1 Comparison Value2. The values to be compared can be any combination of variables (any size), constants, or expressions.

NOTE: On the BS1, expressions are not allowed as arguments. Also, the Value1 (to the left of comparison) must be a variable.

The following example is an IF…THEN command with a simple condition:

1 All 2

A SIMPLE FORM OF IF…THEN

```
IF value < 4000 THEN Main
```

This code will compare the value of *value* to the number 4000. If *value* is less than 4000, the condition is true and the program will jump (implied

```
 Test:
  IF -99 < 100 THEN Is_Less
  DEBUG "Greater than or equal to 100"
  END

Is_Less:
  DEBUG "Less than 100"
  END
```

Although –99 is obviously less than 100, the program will say it is greater. The problem is that –99 is internally represented as the two's complement value 65437, which (using unsigned math) is greater than 100. This phenomena will occur whether or not the negative value is a constant, variable or expression.

LOGICAL OPERATORS (NOT, AND, OR AND XOR).

IF...THEN supports the conditional logic operators NOT, AND, OR, and XOR to allow for more sophisticated conditions, such as multi-part conditions. See Table 5.38 for a list of the operators and Table 5.40 for their effects.

NOTE: The NOT and XOR operators are not available on the BS1.

The NOT operator inverts the outcome of a condition, changing false to true, and true to false. The following IF...THENs are equivalent:

```
IF x <> 100 THEN Not_Equal
IF NOT x = 100 THEN Not_Equal
```

The operators AND, OR, and XOR can be used to join the results of two conditions to produce a single true/false result. AND and OR work the same as they do in everyday speech. Run the example below once with AND (as shown) and again, substituting OR for AND:

NOTE: For BS1's, change lines1 and 2 to:
SYMBOL value1 = B2
SYMBOL value2 = B3

```
value1          VAR     Byte
value2          VAR     Byte

Setup:
  value1 = 5
  value2 = 9

Main:
  IF value1 = 5 AND value2 = 10 THEN Is_True
  DEBUG "Statement is False"
  END

Is_True:
  DEBUG "Statement is True"
  END
```

```
' {$STAMP BS2}
' {$PBASIC 2.0}


sample          VAR     Word            ' Random number to be tested
samps           VAR     Nib             ' Number of samples taken
temp            VAR     Nib             ' Temporary workspace


Setup:
  sample = 11500

Mult3:
  RANDOM sample                         ' Put a random number into sample
  temp = sample // 3
  IF temp <> 0 THEN Mult3               ' Not multiple of 3? -- try again
    DEBUG DEC5 sample, " divides by 3", CR
    samps = samps + 1                   ' Count multiples of 3
    IF samps = 10 THEN Done             ' Quit with 10 samples
  GOTO Mult3                            ' keep checking

Done:
  DEBUG CR, "All done."
  END
```

All 2

Demo Program (IF-THEN-ELSE.bs2)

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

```
' IF-THEN-ELSE.bs2
' The program below generates a series of 16-bit random numbers and tests
' each to determine whether they're evenly divisible by 3. If a number is
' evenly divisible by 3, then it is printed, otherwise, the program
' generates another random number. The program counts how many numbers it
' prints, and quits when this number reaches 10.

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

sample          VAR     Word            ' Random number to be tested
hits            VAR     Nib             ' Number of hits
misses          VAR     Word            ' Number of misses


Setup:
  sample = 11500

Main:
  DO
    RANDOM sample                       ' Put a random number into sample
    IF ((sample // 3) = 0) THEN         ' divisible by 3?
      DEBUG DEC5 sample,                ' - yes, print value and message
            " is divisible by 3", CR
```

```
      BIN1 IN7, CR

OUT7 = 0                              ' Write 0 to output latch
DEBUG "After 0 written to OUT7: ",
      BIN1 IN7, CR

OUTPUT 7                              ' Make P7 an output
DEBUG "After P7 changed to output: ",
      BIN1 IN7
```

location 0. Figure 5.15 indicates the position numbers for characters on the LCD screen.

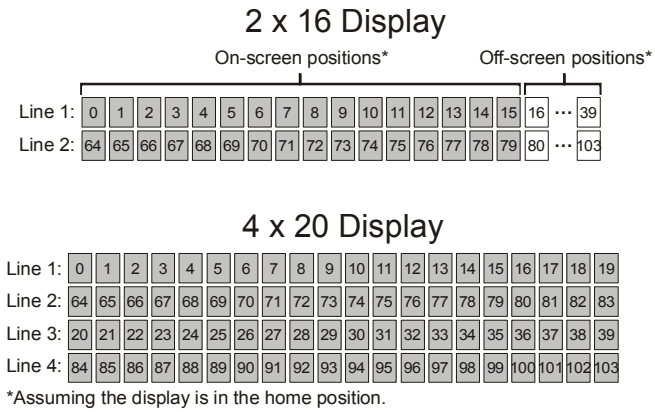
Table 5.45: All LCD Commands (for advanced users). These are supported by LCDs with the Hitachi 44780 controller.

	Command Code (in binary)								Description
	7	6	5	4	3	2	1	0	
Clear Display	0	0	0	0	0	0	0	1	Clear entire display and move cursor home (address 0).
Home Display	0	0	0	0	0	0	1	0	Move cursor home and return display to home position.
Entry Mode	0	0	0	0	0	1	M	S	Sets cursor direction (M: 0=left, 1=right) and display scrolling (S: 0=no scroll, 1=scroll)
Display/Cursor	0	0	0	0	1	D	U	B	Sets display on/off (D), underline cursor (U) and blinking block cursor (B). (0=off, 1=on)
Scroll Display / Shift Cursor	0	0	0	1	C	M	0	0	Shifts display or cursor (C: 0=cursor, 1=display) left or right (M: 0=left, 1=right).
Function Set	0	0	1	B	L	F	0	0	Sets buss size (B: 0=4-bits, 1=8-bits), number of lines (L: 0=1-line, 1=2-lines) and font size (F: 0=5x8, 1=5x10)
Move To CGRAM Address	0	1	A	A	A	A	A	A	Move pointer to character RAM location specified by address (A)
Move To DDRAM Address	1	A	A	A	A	A	A	A	Move cursor to display RAM location specified by address (A)

Note that Figure 5.15 shows the most common DDRAM mapping, though some LCD's may have organized the DDRAM differently. A little experimentation with your LCD may reveal this.

Figure 5.15: LCD Character Positions.

NOTE: Many 1 x 16 displays conform to the position numbers shown on Line 1 of the 2 x 16 display.



LOOKUP

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

1 **LOOKUP *Index*, (*Value0, Value1, …ValueN*), *Variable***

All 2 **LOOKUP *Index*, [*Value0, Value1, …ValueN*], *Variable***

Function

Find the value at location *Index* and store it in *Variable*. If *Index* exceeds the highest index value of the items in the list, *Variable* is left unaffected.

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

- ***Index*** is a variable/constant/expression (0 – 255) indicating the list item to retrieve.
- ***Values*** are variables/constants/expressions (0 – 65535).
- ***Variable*** is a variable that will be set to the value at the *Index* location. If *Index* exceeds the highest location number, *Variable* is left unaffected.

Quick Facts

Table 5.55: LOOKUP Quick Facts.

	BS1 and all BS2 Models
Limit of *Value* Entries	256
Starting Index Number	0
If index exceeds the highest location…	Variable is left unaffected
Related Command	LOOKDOWN

Explanation

LOOKUP retrieves an item from a list based on the item's position, *Index,* in the list. For example:

1
```
SYMBOL  index   = B2
SYMBOL  result  = B3

index = 3
result = 255

LOOKUP index, (26, 177, 13, 1, 0, 17, 99), result
DEBUG  "Item ", #index, "is: ", #result
```

-- or --

Demo Program (LOW.bs2)

```
' LOW.bs2
' This simple program sets I/O pin 0 high for 1/2 second and low for
' 1/2 second in an endless loop.  Connect an LED to P0 for a simple
' blinker.

' {$STAMP BS2}


Main:
  HIGH 0
  PAUSE 500
  LOW 0
  PAUSE 500
  GOTO Main
  END
```

NOTE: This example program can be used with the BS1 and all BS2 models by changing the $STAMP directive accordingly.

the actual timing to vary by as much as –50, +100 percent (i.e., a *Duration* of 0, NAP can range from 9 to 36 ms). At room temperature with a fresh battery or other stable power supply, variations in the length of a NAP will be less than ±10 percent.

A GREAT USE FOR NAP; FREE POWER SAVINGS.

One great use for NAP is in a battery-powered application where at least a small amount of time is spent doing nothing. For example, you may have a program that loops endlessly, performing some task, that pauses for approximately 100 ms each time through the loop. You could replace your PAUSE 100 with NAP 3, as long as the timing of the 100 ms pause was not critical. The NAP 3 would effectively pause your program for about 144 ms and, at the same time, would place the BASIC Stamp in low-power mode, which would extend your battery life.

TIPS FOR DRIVING LOADS DURING NAP.

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

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

The demo program can be used to demonstrate the effects of the NAP glitch with an LED and resistor as shown in Figure 5.18.

ON

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

NOTE: ON requires PBASIC 2.5.

All 2 **ON *Offset* GOTO *Address1, Address2, …AddressN***

All 2 **ON *Offset* GOSUB *Address1, Address2, …AddressN***

Function

GOTO or GOSUB to the *Address* specified by *Offset* (if in range). ON is similar in operation to BRANCH with the exception that program execution can optionally return to the line following ON (if using ON…GOSUB).

- ***Offset*** is a variable/constant/expression (0 - 255) that specifies the index (0 - N) of the address, in the list, to GOTO or GOSUB to.
- ***Address*** is a label that specifies where to go for a given *Offset*. ON will ignore any list entries beyond offset 255.

Quick Facts

Table 5.61: ON Quick Facts.

	All BS2 Models
Limit of Address Entries	256
Maximum GOSUBs per Program	255 (each ON…GOSUB counts as one GOSUB, regardless of number of address list entries)
Maximum Nested GOSUBS	4
Related Commands	BRANCH, GOTO and GOSUB

Explanation

The ON instruction is like saying, "Based ON the value of *Offset*, GOTO or GOSUB to one of these *Addresses."* ON is useful when you want to write something like this:

```
IF (value = 0) THEN GOTO Case_0        ' "GOTO" jump table
IF (value = 1) THEN GOTO Case_1
IF (value = 2) THEN GOTO Case_2
```

- or -

```
IF (value = 0) THEN GOSUB Case_0       ' "GOSUB" jump table
IF (value = 1) THEN GOSUB Case_1
IF (value = 2) THEN GOSUB Case_2
```

OWOUT

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

OWOUT ***Pin, Mode,*** **[** ***OutputData*** **]**

Function

Send data to a device using the 1-Wire protocol.

- ***Pin*** is a variable/constant/expression (0 – 15) that specifies which I/O pin to use. 1-Wire devices require only one I/O pin (called DQ) to communicate. This I/O pin will be toggled between output and input mode during the OWOUT command and will be set to input mode by the end of the OWOUT command.
- ***Mode*** is a variable/constant/expression (0 – 15) indicating the mode of data transfer. The *Mode* argument controls placement of reset pulses (and detection of presence pulses) as well as byte vs. bit input and normal vs. high speed. See explanation below.
- ***OutputData*** is a list of variables and modifiers that tells OWOUT how to format outgoing data. OWOUT can transmit individual or repeating bytes, convert values into decimal, hexadecimal or binary text representations, or transmit strings of bytes from variable arrays. These actions can be combined in any order in the *OutputData* list.

Quick Facts

Table 5.68: OWOUT Quick Facts.

	BS2p, BS2pe, and BS2px
Transmission Rate	Approximately 20 kbits/sec (low speed, not including reset pulse)
Special Notes	The DQ pin (specified by *Pin*) must have a 4.7 KΩ pull-up resistor. The BS2pe is not capable of high-speed transfers.
Related Command	OWIN

Explanation

The 1-Wire protocol is a form of asynchronous serial communication developed by Dallas Semiconductor. It only requires one I/O pin and that pin can be shared between multiple 1-Wire devices. The OWOUT command allows the BASIC Stamp to send data to a 1-Wire device.

A SIMPLE OWOUT EXAMPLE.

The following is an example of the OWOUT command:

```
OWOUT  0, 1, [$4E]
```

```
temp    VAR     byte

PUT 25, 100                             ' put low byte
GET 25, temp                            ' read byte value
DEBUG DEC temp                          ' display byte value
```

When using the $PBASIC 2.5 directive, multiple sequential values may be stored to SPRAM, starting at *Location*, and the WORD modifier may be specified to store 16-bit values.

```
' {$PBASIC 2.5}

temp    VAR     Word

PUT 25, Word 2125                       ' write word value
GET 25, Word temp                       ' read word value
DEBUG DEC temp                          ' display 2125
```

Most Scratch Pad RAM locations are available for general use. The highest locations have a special, read-only purpose; see the GET command for more information.

SCRATCH PAD RAM LOCATIONS AND THEIR PURPOSE.

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}

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

value           VAR     Byte
idx             VAR     Byte


Setup:
  GET Slot, value
```

A SIMPLE SHIFTOUT EXAMPLE.

Here is a simple example:

```
SHIFTOUT 0, 1, MSBFIRST, [250]
```

Here, the SHIFTOUT command will write to I/O pin 0 (*Dpin*) and will generate a clock signal on I/O pin 1 (*Cpin*). The SHIFTOUT command will generate eight clock pulses while writing each bit (of the 8-bit value 250) onto the data pin (*Dpin*). In this case, it will start with the most significant bit first as indicated by the *Mode* value of MSBFIRST.

CONTROLLING THE NUMBER OF BITS TRANSMITTED.

By default, SHIFTOUT transmits eight bits, but you can set it to shift any number of bits from 1 to 16 with the *Bits* argument. For example:

```
SHIFTOUT 0, 1, MSBFIRST, [250\4]
```

Will output only the lowest (rightmost) four bits (%1010 in this case). But what if you want to output the leftmost bits of a given value? By adding the right-shift operator (>>) to the code you can adjust the output as required:

```
SHIFTOUT 0, 1, MSBFIRST, [(250 >> 2)\6]
```

will output the upper six bits (%111110 in this case).

Some devices require more than 16 bits. To solve this, you can use a single SHIFTOUT command with multiple values. Each value can be assigned a particular number of bits with the *Bits* argument. As in:

```
SHIFTOUT 0, 1, MSBFIRST, [250\4, 1045\16]
```

The above code will first shift out four bits of the number 250 (%1010) and then 16 bits of the number 1045 (%0000010000010101). The two values together make up a 20 bit value.

SHIFTOUT ACCEPTS VARIABLES AND EXPRESSIONS FOR *OUTPUTDATA* AND *BITS* ARGUMENTS.

In the examples above, specific numbers were entered as the data to transmit, but, of course, the SHIFTOUT command will accept variables and expressions for the *OutputData* and even for the *Bits* argument.

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).

guarantee that the state actually changes, regardless of the initial input or output mode, do this:

```
PIN2 = PIN2                             ' make output driver match input
TOGGLE 2                                ' then toggle
```

- or -

```
OUT2 = IN2                              ' make output driver match input
TOGGLE 2                                ' then toggle
```

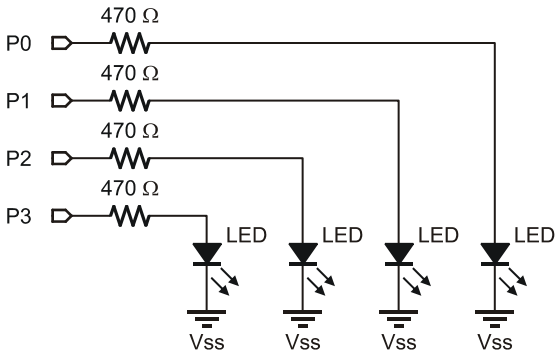


Figure 5.47: Example LED Circuit for TOGGLE Demo Programs.

Demo Program (TOGGLE.bs1)

```
' TOGGLE.bs1
' Connect LEDs to pins 0 through 3 as shown in the TOGGLE command descrip-
' tion in the manual and run this program. The TOGGLE command will treat
' you to a light show. You may also run the demo without LEDs. The Debug
' window will show you the states of pins 0 through 3.

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

SYMBOL  thePin          = B0            ' pin 0 - 3

Setup:
  DIRS = %1111                          ' make LEDs output, low

Main:
  FOR thePin = 0 TO 3                   ' loop through pins
    TOGGLE thePin                       ' toggle current pin
    DEBUG CLS, %PINS                    ' show on Debug
    PAUSE 100                           ' short delay
  NEXT
  GOTO Main                             ' repeat forever
  END
```