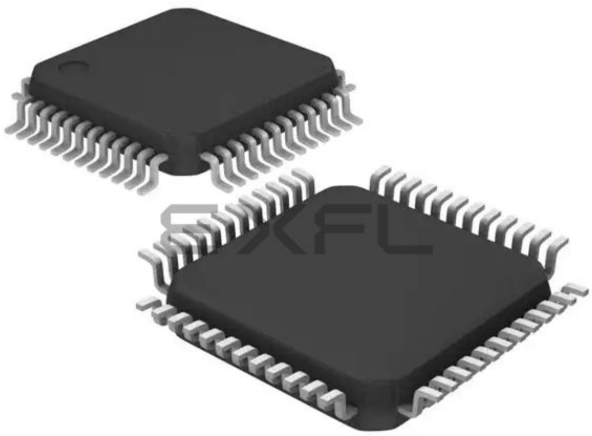

Welcome to E-XFL.COM

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-p40

Basic Stamp 2px

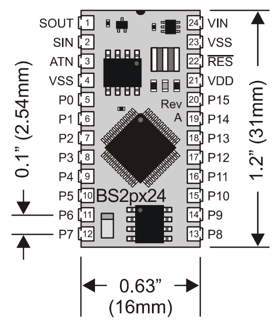

Figure 1.12: BASIC Stamp 2px (Rev. A) (Stock# BS2px-IC)

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

Table 1.7: BASIC Stamp 2px 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.

EASY STEPS TO CREATING MULTI-FILE PROJECTS.

To create a project consisting of multiple files, follow these steps:

1. Create the first file in the editor and save it (we'll call it Sample.bsx). This will be the program that is downloaded into program slot 0.
2. Create at least one other file in the editor and save it also (we'll call it NextProgram.bsx).

Note: At this point the editor tabs will be:

0:Sample.bsx and 0:NextProgram.bsx.

indicating that there are two unrelated files open "Sample.bsx" and "NextProgram.bsx" and each will be downloaded into program slot 0.

3. Go back to the first program and enter or modify the $STAMP directive using the project format. Use "NextProgram" as the *File2* argument. For example:

```
' {$STAMP  BS2sx,  NextProgram.bsx}
```

4. Then tokenize the code by pressing F7 or selecting Run → Check Syntax from the menu.

At this point, the BASIC Stamp Editor will see the $STAMP directive and realize that this file (Sample.bsx) is the first file in a project and that the second file should be NextProgram.bsx. It will then search for the file on the hard drive (to verify its path is correct), will see that it is already loaded, and then will change the editor tabs to indicate the project relationship. At this point the editor tabs will be:

0:Sample.bsx and [Sample] 1:NextProgram.bsx.

indicating that there are two related files open; "Sample.bsx" and "NextProgram.bsx". NextProgram.bsx belongs to the "Sample" project and it will be downloaded into program slot 1 and Sample.bsx will be downloaded into program slot 0.

```
' {$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:

```
' {$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.

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

The result returned by | will contain 1s in any bit positions in which one or the other (or both) input values contain 1s. Example:

1

```
SYMBOL  value1          = B2
SYMBOL  value2          = B3
SYMBOL  result          = B4

value1 = %00001111
value2 = %10101001
result = value1 | value2
DEBUG %result                               ' Show result of OR (%10101111)
```

-- or --

```
DEBUG  BIN  ?  %00001111  |  %10101001   ' Show result of OR (%10101111)
```

XOR: ^

All 2

The Xor operator (^) returns the bitwise XOR of two values. Each bit of the values is subject to the following logic:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

The result returned by ^ will contain 1s in any bit positions in which one or the other (but not both) input values contain 1s. Example:

1

```
SYMBOL  value1          = B2
SYMBOL  value2          = B3
SYMBOL  result          = B4

value1 = %00001111
value2 = %10101001
result = value1 ^ value2
DEBUG %result                             ' Show result or XOR (%10100110)
```

-- or --

AUXIO

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

2p 2pe 2px **AUXIO**

Function

Switch from control of main I/O pins to auxiliary I/O pins (on the BS2p40 only).

Quick Facts

Table 5.2: AUXIO Quick Facts.

	BS2p, BS2pe, and BS2px
I/O pin IDs	0 – 15 (just like main I/O, but after AUXIO command, all references affect physical pins 21 – 36).
Special Notes	The BS2p, BS2pe, and BS2px 24-pin modules accept this command, however, only the BS2p40 gives access to the auxiliary I/O pins.
Related Commands	MAINIO and IOTERM

Explanation

The BS2p, BS2pe, and BS2px are available as 24-pin modules that are pin compatible with the BS2, BS2e and BS2sx. Also available is a 40-pin module called the BS2p40, with an additional 16 I/O pins (for a total of 32). The BS2p40's extra, or auxiliary, I/O pins can be accessed in the same manner as the main I/O pins (by using the IDs 0 to 15) but only after issuing an AUXIO or IOTERM command. The AUXIO command causes the BASIC Stamp to affect the auxiliary I/O pins instead of the main I/O pins in all further code until the MAINIO or IOTERM command is reached, or the BASIC Stamp is reset or power-cycled. AUXIO is also used when setting the DIRS register for auxiliary I/O pins on the BS2p40.

When the BASIC Stamp module is reset, all RAM variables including DIRS and OUTS are cleared to zero. This affects both main and auxiliary I/O pins. On the BS2p24, BS2pe, and BS2px, the auxiliary I/O pins from the interpreter chip are not connected to physical I/O pins on the BASIC Stamp module. While not connected to anything, these pins do have internal pull-up resistors activated, effectively connecting them to Vdd. After reset, reading the auxiliary I/O from a BS2p24, BS2pe24, or BS2px24 will return all 1s.

```
    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
```

To retrieve a word-sized value, you'll need to use the WORD modifier in the READ command and a word-sized variable.

Finally, a *DataItem* may be defined using a simple expression with the binary operators shown in Table 4.5. For example,

```
MinLvl          CON     10

myLvl           VAR     Byte

Level1          DATA    MinLvl + 10
Level2          DATA    MinLvl * 5 + 21

READ Level2, myLvl                     ' read EE location Level2
DEBUG DEC myLvl                        ' show value of myLvl (71)
```

All 2

Demo Program (DATA.bs2)

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

```
' DATA.bs2
' This program stores a number of large text strings into EEPROM with the
' DATA directive and then sends them, one character at a time via the DEBUG
' command.  This is a good demonstration of how to save program space by
' storing large amounts of data in EEPROM directly, rather than embedding
' the data into DEBUG commands.

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

idx         VAR     Word            ' current location number
phrase      VAR     Nib             ' current phrase number
char        VAR     Byte            ' character to print

' ----- Define all text phrases (out of order, just for fun!) -----
'
Text1       DATA    "Here is the first part of a large chunk of textual "
            DATA    "data ", CR, "that needs to be transmitted.  There's "
            DATA    "a 5 second delay", CR, "between text paragraphs. ", CR
            DATA    CR, 0

Text3       DATA    "The alternative (having multiple DEBUGs or SEROUTs, "
            DATA    "each ", CR, "with their own line of text) consumes "
            DATA    "MUCH more EEPROM ", CR, "(program) space. ", CR
            DATA    CR, 0

Text6       DATA    "The 0 is used by this program to indicate we've "
            DATA    "reached the ", CR, "End of Text.  The Main routine "
            DATA    "pauses in between each block of", CR, "text,and then "
            DATA    "uses a LOOKUP command to retrieve the location ", CR
            DATA    "of the next desired block of text to print. ", 0
```

GET

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

2e 2sx 2p 2pe 2px **GET** ***Location*, { WORD } *Variable* { , { WORD } *Variable...* }**

NOTE: The optional arguments require PBASIC 2.5.

Function

Read the value from Scratch Pad RAM (SPRAM) *Location* and store in *Variable*.

- ***Location*** is a variable/constant/expression (0 – 63 for BS2e and BS2sx and 0 – 131 for BS2p, BS2pe, and BS2px) that specifies the SPRAM location to read from.
- ***Variable*** is a variable (usually a byte, or word if using the optional WORD modifier) in which to store the value.

Quick Facts

Table 5.27: GET Quick Facts.

	BS2e and BS2sx	BS2p, BS2pe, and BS2px
Scratch Pad RAM Size and Organization	64 bytes (0 – 63). Organized as bytes only.	136 bytes (0 – 135). Organized as bytes only.
General Purpose Locations	0 - 62	0 – 126
Special Use Location	Location 63: Active program slot number (read only).	Location 127: READ/WRITE slot and Active Program slot (read only). Locations 128-135: Polled Interrupt status (read only).
Related Commands	PUT	PUT and STORE, and SPSTR formatter.
PBASIC 2.5 Syntax Options	Multiple sequential variables may be read from the Scratch Pad RAM. The optional WORD modifier may be specified to retrieve 16-bit values.	

Explanation

The GET command reads a value from the specified Scratch Pad RAM location and stores it into *Variable*. All values in all locations can be retrieved from within any of the 8 program slots.

USES FOR SCRATCH PAD RAM.

SPRAM is useful for passing data to programs in other program slots and for additional workspace. It is different than regular RAM in that symbol names cannot be assigned directly to locations and each location is always configured as a byte only. The following code will read the value at location 25, store it in a variable called *temp* and display it:

All 2

Demo Program (ON-GOSUB.bs2)

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

```
' ON-GOSUB.bs2
' This program demonstrates a simple task manager that can be used
' in a variety of applications.  It is particularly useful in
' robotics and industrial applications.  The advantage of this
' design is that task code modules may be called from other places
' in the program, including other tasks, and the overall program flow
' is maintained.

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

task            VAR     Nib


Main:
  DO
    ON task GOSUB Task_0, Task_1, Task_2        ' run current task
    task = task + 1 // 3                        ' update task pointer
    PAUSE 1000
  LOOP
  END

Task_0:
  DEBUG "Running Task 0", CR
  RETURN

Task_1:
  DEBUG "Running Task 1", CR
  RETURN

Task_2:
  DEBUG "Running Task 2", CR
  RETURN
```

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]
```

PAUSE

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

1 All 2

PAUSE *Duration*

Function

1

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

Pause the program (do nothing) for the specified *Duration*.

- ***Duration*** is a variable/constant/expression (0 – 65535) that specifies the duration of the pause. The unit of time for *Duration* is 1 millisecond.

Explanation

PAUSE delays the execution of the next program instruction for the specified number of milliseconds. For example:

```
Flash:
  LOW 0
  PAUSE 100
  HIGH 0
  PAUSE 100
  GOTO Flash
```

This code causes pin 0 to go low for 100 ms, then high for 100 ms. The delays produced by PAUSE are as accurate as the ceramic-resonator time base (on the BASIC Stamp modules), ±1 percent. When you use PAUSE in timing-critical applications, keep in mind the relatively low speed of the PBASIC interpreter. This is the time required for the BASIC Stamp to read and interpret an instruction stored in the EEPROM.

1 All 2

Demo Program (PAUSE.bs2)

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

```
' PAUSE.bs2
' This program demonstrates the PAUSE command's time delays. Once a second,
' the program will put the message "Paused..." on the screen.
' {$STAMP BS2}

Main:
  DEBUG "Paused...", CR
  PAUSE 1000
  GOTO Main
```

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

use this formula: (*Duty*/255) * 5V. For example, if *Duty* is 100, (100/255) * 5V = 1.96V; PWM outputs a train of pulses whose average voltage is 1.96V.

FILTERING THE PWM SIGNAL.

In order to convert PWM into an analog voltage we have to filter out the pulses and store the average voltage. The resistor/capacitor combination in Figure 5.31 will do the job. The capacitor will hold the voltage set by PWM even after the instruction has finished. How long it will hold the voltage depends on how much current is drawn from it by external circuitry, and the internal leakage of the capacitor. In order to hold the voltage relatively steady, a program must periodically repeat the PWM instruction to give the capacitor a fresh charge.

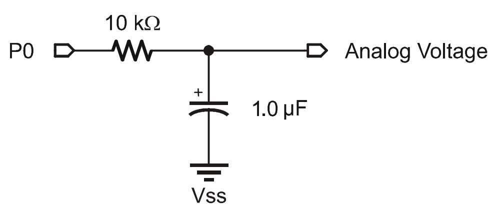

Figure 5.31: Example PWM Filter Circuit.

DETERMINING THE APPROPRIATE CYCLE TIME FOR YOUR CIRCUIT.

Just as it takes time to discharge a capacitor, it also takes time to charge it in the first place. The PWM command lets you specify the charging time in terms of PWM cycles. The period of each cycle is shown in Table 5.86. So, on the BS2, to charge a capacitor for 5ms, you would specify 5 cycles in the PWM instruction.

How do you determine how long to charge a capacitor? Use this rule-of-thumb formula: Charge time = 5 * R * C. For instance, Figure 5.31 uses a 10 kΩ (10×10^3 ohm) resistor and a 1 µF (1×10^{-6} F) capacitor:

Charge time = $5 * 10 \times 10^3 * 1 \times 10^{-6} = 50 \times 10^{-3}$ seconds, or 50 ms.

Since, on the BS2, each cycle is approximately a millisecond, it would take at least 50 cycles to charge the capacitor. Assuming the circuit is connected to pin 0, here's the complete PWM instruction:

```
PWM 0, 100, 50                    ' charge to 1.96 V
```

After outputting the PWM pulses, the BASIC Stamp leaves the pin in input mode (0 in the corresponding bit of DIRS). In input mode, the pin's output driver is effectively disconnected. If it were not, the steady output

```
' msb first so that the msb appears on pin QH and the lsb on QA. Changing
' MSBFIRST to LSBFIRST causes the data to appear backwards on the outputs.

Main:
  DO
    SHIFTOUT Dpin, Clk, MSBFIRST, [counter]     ' send the bits
    PULSOUT Latch, 1                            ' transfer to outputs
    PAUSE 100                                   ' Wait 0.1 seconds
    counter = counter + 1                       ' increment counter
  LOOP
  END
```

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
```

All 2

Demo Program (TOGGLE.bs2)

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

```
' TOGGLE.bs2
' 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.

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

thePin          VAR     Nib             ' pin 0 - 3


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

Main:
  DO
    FOR thePin = 0 TO 3                 ' loop through pins
      TOGGLE thePin                     ' toggle current pin
      DEBUG HOME, BIN4 OUTA             ' show on Debug
      PAUSE 250                         ' short delay
    NEXT
  LOOP                                  ' repeat forever
  END
```

Reserved Words

This appendix contains complete listings of the reserved words for PBASIC 1.0, PBASIC 2.0, and PBASIC 2.5, current with the BASIC Stamp Editor v2.1.

The reserved word lists have been organized into 4 tables, because it varies with each BASIC Stamp model and version of PBASIC. Table B.1 shows the reserved words for the BASIC Stamp 1, using the required PBASIC 1.0.

Table B.1: BS1 Reserved Words.

BS1				
AND	GOSUB	N2400	PIN0..PIN7	SOUND
B0..B13	GOTO	NAP	PINS	STEP
BIT0..BIT15	HIGH	NEXT	PORT	SYMBOL
BRANCH	IF	ON300	POT	T300
BSAVE	INPUT	ON600	PULSIN	T600
BUTTON	LET	ON1200	PULSOUT	T1200
CLS	LOOKDOWN	ON2400	PWM	T2400
CR	LOOKUP	OR	RANDOM	THEN
DEBUG	LOW	OT300	READ	TO
DIR0..DIR7	MAX	OT600	RETURN	TOGGLE
DIRS	MIN	OT1200	REVERSE	W0..W6
EEPROM	N300	OT2400	SERIN	WRITE
END	N600	OUTPUT	SEROUT	
FOR	N1200	PAUSE	SLEEP	

Table B.2 on the following page lists the reserved words common to all BS2 models, including those for PBASIC 2.0 and PBASIC 2.5. Words listed that are only reserved when using PBASIC 2.5 are marked with ($^{2.5}$).