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	Active
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/pbasic1xt-ss

BASIC Stamp 1

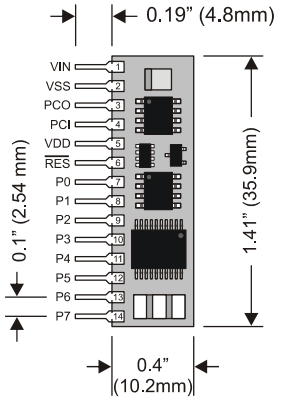


Figure 1.1: BASIC Stamp 1 (Rev B) (Stock# BS1-IC).

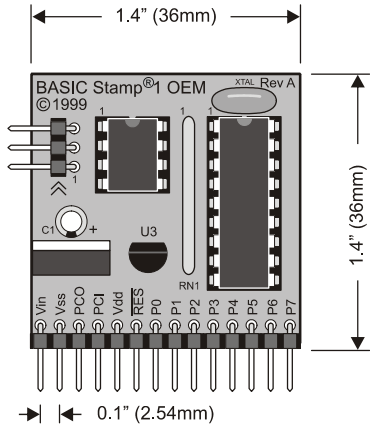
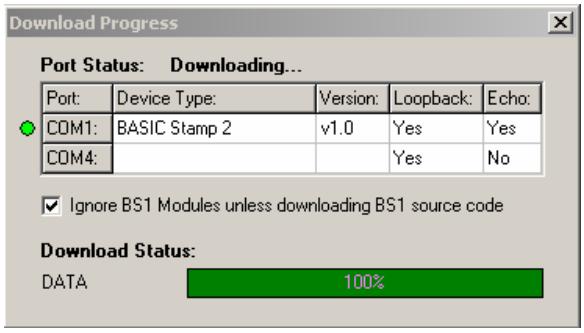


Figure 1.2: BASIC Stamp 1 OEM (Rev. A) (Stock# 27295).

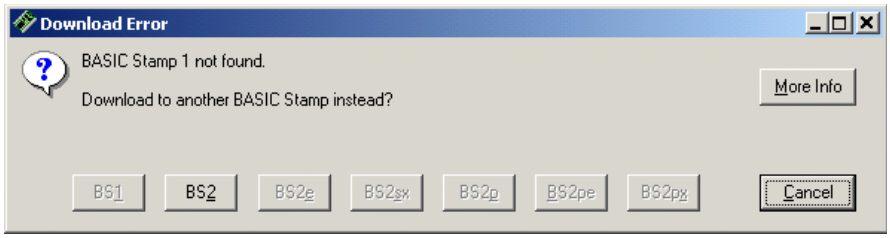
to the BASIC Stamp module (assuming the code is correct and the BASIC Stamp is properly connected). The Download Progress window looks similar to the Identify window with the exception of the additional Download Status progress bar, and the indicator LED by the port transmitting the data.

Figure 3.11: The Download Progress Window.



If any errors occur, such as communication failure or inability to detect a BASIC Stamp module, you will be prompted appropriately. One possible error occurs when the BASIC Stamp your PBASIC program is targeting does not appear to be connected to the PC (see Figure 3.12). This may be caused, for example, by opening up a BASIC Stamp 1 program (usually has a .bas or .bs1 extension) and trying to download it to a BASIC Stamp 2 module, instead.

Figure 3.12: A Download Error message.



When this happens, you'll be prompted to correct the situation, quickly done by clicking on the BS2 button (if you really intended to download to the BS2 in the first place). Keep in mind that programs written for one BASIC Stamp model may not function properly on a different BASIC Stamp model. Click on the More Info button for more detail. NOTE: If you select the BS2 button, as in this example, the editor will modify the

expression 100+90 into a compile-time expression like OneNinety CON 100+90.

To sum up: compile-time expressions are those that involve only constants; once a variable is involved, the expression must be solved at run-time. That's why the line "NotWorking CON 3 * result" would generate an error message. The CON directive works only at compile-time and *result* is a variable; variables are not allowed in compile-time expressions.

Defining and Using Pins with the PIN directive.

All 2

Now we know now to create variables and constants (with VAR and CON) but there is a third option if you're using PBASIC 2.5; pin-type symbols (with PIN). PIN is like VAR and CON put together and represents an I/O pin.

There are some situations where it is handy to refer to a pin using a variable (like IN2 or OUT2) and also as a constant (2, in this case). The PIN directive lets you define a context-sensitive symbol representing an I/O pin. Depending on where and how this pin-type symbol is used determines whether it is treated as an I/O pin input variable, and I/O pin output variable or as a constant representing the pin number.

Let's explore a simple example to see where this is useful. It is common practice to define constants for any number used in many places so that changing that number doesn't create a maintenance hassle later on. If we were to use a constant symbol to represent an I/O pin, we might do something like this:

```
' {$PBASIC 2.5}

signal    CON   1              ' constant-type symbol representing I/O 1

INPUT  signal                  ' set signal pin to input

Wait:
  IF signal = 0 THEN Wait      ' wait until signal pin = 1
```

Here we define *signal* to represent our desired I/O pin, then we use the INPUT command to set it to the input direction and later we check the state of the signal pin and loop (wait) while it is equal to logic 0. This code has a common bug, however; the INPUT command works as expected, because its *Pin* argument requires a number representing the I/O pin, but

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.

```
DEBUG BIN8 ? %00001111 ^ %10101001     ' Show result of XOR (%10100110)
```

All 2

The And Not operator (&/) returns the bitwise AND NOT of two values. Each bit of the values is subject to the following logic:

AND NOT: &/

1

0 AND NOT 0 = 0
0 AND NOT 1 = 0
1 AND NOT 0 = 1
1 AND NOT 1 = 0

The result returned by &/ will contain 1s in any bit positions in which the first value is 1 and the second value is 0. Example:

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

value1 = %00001111
value2 = %10101001
result = value1 &/ value2
DEBUG %result                       ' Show result of AND NOT (%00000110)
```

1

The Or Not operator (|/) returns the bitwise OR NOT of two values. Each bit of the values is subject to the following logic:

OR NOT: |/

1

0 OR NOT 0 = 1
0 OR NOT 1 = 0
1 OR NOT 0 = 1
1 OR NOT 1 = 1

The result returned by |/ will contain 1s in any bit positions in which the first value is 1 or the second value is 0. Example:

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

value1 = %00001111
value2 = %10101001
result = value1 |/ value2
DEBUG %result                        ' Show result of OR NOT (%01011111)
```

1

XOR NOT: ^/

The Xor Not operator (^/) returns the bitwise XOR NOT of two values. Each bit of the values is subject to the following logic:

0 XOR NOT 0 = 1
0 XOR NOT 1 = 0
1 XOR NOT 0 = 0
1 XOR NOT 1 = 1

The result returned by ^/ will contain 1s in any bit positions in which the first value and second values are equal.

Example:

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

value1 = %00001111
value2 = %10101001
result = value1 ^/ value2
DEBUG %result                           ' Show result of OR NOT (%01011001)
```

```
  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
```

After running the above code, "x = $4B" and "x = %01001011" should appear on the screen. To display hexadecimal or binary values without the "symbol = " preface, use the value formatter (#) before the $ and %, as shown below:

```
SYMBOL  x = B2

x = 75
DEBUG #x, "as HEX is ", #$x       ' displays "75 as HEX is $4B"
DEBUG #x, "as BINARY is ", #%x    ' displays "75 as BINARY is %01001011"
```

DISPLAYING ASCII CHARACTERS (BS1).

To display a number as its ASCII character equivalent, use the ASCII formatter (@).

```
SYMBOL  x = B2

x = 75
DEBUG @x
```

Table 5.10: DEBUG Formatters for the BASIC Stamp 1.

Formatter	Description
#	Suppresses the "symbol = x" format and displays only the 'x' value. The default format is decimal but may be combined with any of the formatters below (ex: #x to display: x value)
@	Displays "symbol = 'x'" + carriage return; where x is an ASCII character.
$	Hexadecimal text.
%	Binary text.

USING CR AND CLS (BS1).

Two pre-defined symbols, CR and CLS, can be used to send a carriage-return or clear-screen command to the Debug Terminal. The CR symbol will cause the Debug Terminal to start a new line and the CLS symbol will cause the Debug Terminal to clear itself and place the cursor at the top-left corner of the screen. The following code demonstrates this.

```
DEBUG  "You can not see this.", CLS, "Here is line 1", CR, "Here is line 2"
```

When the above is run, the final result is "Here is line 1" on the first line of the screen and "Here is line 2" on the second line. You may or may not have seen "You can not see this." appear first. This is because it was immediately followed by a clear-screen symbol, CLS, which caused the display to clear the screen before displaying the rest of the information.

NOTE: The rest of this discussion does not apply to the BASIC Stamp 1.

```
row             VAR     Nib

Setup:
  col = 0

Main:
  DO WHILE (col < 10)                   ' attempt 10 iterations
    FOR row = 0 TO 15                   ' attempt 16 iterations
      IF (row > 9) THEN EXIT            ' terminate when row > 9
      DEBUG CRSRXY, (col * 8), row,     ' print col/row at location
            DEC col, "/", DEC row, CR
    NEXT
    col = col + 1                       ' update column
    IF (col = 3) THEN EXIT              ' terminate when col = 3
  LOOP
  END
```

I2CIN

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

I2CIN ***Pin, SlaveID,*** **{** ***Address*** **{ *****LowAddress*** **},** **}** **[** ***InputData*** **]**

Function

Receive data from a device using the I²C protocol.

- ***Pin*** is a variable/constant/expression (0 or 8) that specifies which I/O pins to use. I²C devices require two I/O pins to communicate. The *Pin* argument serves a double purpose; specifying the first pin (for connection to the chip's SDA pin) and, indirectly, the other required pin (for connection to the chip's SCL pin). See explanation below. Both I/O pins will be toggled between output and input mode during the I2CIN command and both will be set to input mode by the end of the I2CIN command.
- ***SlaveID*** is a variable/constant/expression (0 – 255) indicating the unique ID of the I²C chip.
- ***Address*** is an optional variable/constant/expression (0 – 255) indicating the desired address within the I²C chip to receive data from. The *Address* argument may be used with the optional *LowAddress* argument to indicate a word-sized address value.
- ***LowAddress*** is an optional variable/constant/expression (0 – 255) indicating the low-byte of the word-sized address within the I²C chip to receive data from. This argument must be used along with the *Address* argument.
- ***InputData*** is a list of variables and modifiers that tells I2CIN what to do with incoming data. I2CIN can store data in a variable or array, interpret numeric text (decimal, binary, or hex) and store the corresponding value in a variable, wait for a fixed or variable sequence of bytes, or ignore a specified number of bytes. These actions can be combined in any order in the *InputData* list.

```
addr            VAR     Word            ' internal address
block           VAR     Nib             ' block address in 24LC16
value           VAR     Byte            ' value to write
check           VAR     Nib             ' for checking retuned values
result          VAR     Byte(16)        ' array for returned value


Write_To_EEPROM:
  DEBUG "Writing...", CR
  PAUSE 2000
  FOR addr = 0 TO 2047 STEP 16          ' loop through all addresses
    block = addr.NIB2 << 1              ' calculate block address
    value = addr >> 4                   ' create value from upper 8 bits
    ' write 16 bytes
    I2COUT SDA, $A0 | block, addr, [REP value\16]
    PAUSE 5
    DEBUG "Addr: ", DEC4 addr, "-", DEC4 addr + 15, "  ",
          "Value: ", DEC3 value, CR
  NEXT
  PAUSE 2000

Read_From_EEPROM:
  DEBUG CR, "Reading...", CR
  PAUSE  2000
  FOR addr = 0 TO 2047 STEP 16
    block = addr.NIB2 << 1
    value = addr >> 4
    I2CIN SDA, $A1 | block, addr, [STR result\16]
    FOR check = 0 TO 15
      IF (result(check) <> value) THEN Error
    NEXT
    DEBUG "Addr: ", DEC4 addr, "-", DEC4 addr + 15, "  ",
          "Value: ", DEC3 result, CR
  NEXT
  PAUSE 100
  DEBUG CR, "All locations passed"
  END


Error:
  DEBUG "Error at location: ", DEC4 addr + check, CR,
        "Found: ", DEC3 result(check), ", Expected: ", DEC3 value
  END
```

```
value     VAR  Byte(13)
LCDIN  0, 128, [value]                'receive the ASCII value for "V"
LCDIN  0, 128, [DEC  value]           'receive the number 3.
LCDIN  0, 128, [HEX  value]           'receive the number $3A.
LCDIN  0, 128, [BIN  value]           'receive the number %101.
LCDIN  0, 128, [STR value\13]         'receive the string "Value: 3A:101"
```

Table 5.47 and Table 5.48 list all the special formatters and conversion formatters available to the LCDIN command. See the SERIN command for additional information and examples of their use.

Some possible uses of the LCDIN command are 1) in combination with the LCDOUT command to store and read data from the unused DDRAM or CGRAM locations (as extra variable space), 2) to verify that the data from a previous LCDOUT command was received and processed properly by the LCD, and 3) to read character data from CGRAM for the purposes of modifying it and storing it as a custom character.

Table 5.47: LCDIN Special Formatters.

Special Formatter	Action
SPSTR L	Input a character string of length L bytes (up to 126) into Scratch Pad RAM, starting at location 0. Use GET to retrieve the characters.
STR *ByteArray* \L {\E}	Input a character string of length L into an array. If specified, an end character E causes the string input to end before reaching length L. Remaining bytes are filled with 0s (zeros).
WAIT (*Value)*	Wait for a sequence of bytes specified by value. Value can be numbers separated by commas or quoted text (ex: 65, 66, 67 or "ABC"). The WAIT formatter is limited to a maximum of six characters.
WAITSTR *ByteArray* {\L}	Wait for a sequence of bytes matching a string stored in an array variable, optionally limited to L characters. If the optional L argument is left off, the end of the array-string must be marked by a byte containing a zero (0).
SKIP *Length*	Ignore *Length* bytes of characters.

\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

Table 5.66: OWIN Conversion Formatters

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.

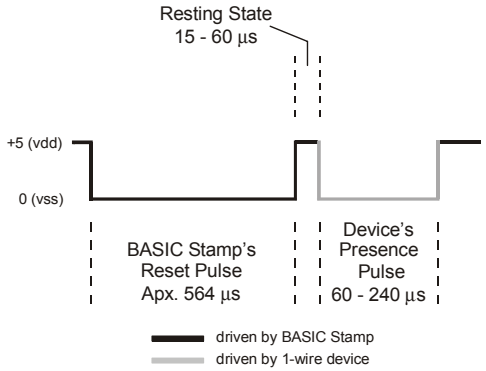
3 The hexadecimal formatters are not case-sensitive; “a” through “f” means the same as “A” through “F”.

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.

Figure 5.24: OWOUT Reset and Presence Pulse.



This reset pulse is controlled by the lowest two bits of the *Mode* argument in the OWOUT command. It can be made to appear before the ROM Function Command (ex: *Mode* = 1), after the Transaction/Data portion (ex: *Mode* = 2), before and after the entire transaction (ex: *Mode* = 3) or not at all (ex: *Mode* = 0). See the section on *Mode*, above, for more information.

Following the Initialization part is the ROM Function Command. The ROM Function Command is used to address the desired 1-Wire device. Table 5.72 shows common ROM Function Commands. If only a single 1-wire device is connected, the Skip ROM command may be used to address it. If more than one 1-wire device is attached, the BASIC Stamp will ultimately have to address them individually using the Match ROM command.

Table 5.72: OWOUT ROM Function Commands.

Command	Value (in Hex)	Action
Read ROM	$33	Reads the 64-bit ID of the 1-Wire device. This command can only be used if there is a single 1-Wire device on the line.
Match ROM	$55	This command, followed by a 64-bit ID, allows the BASIC Stamp to address a specific 1-Wire device.
Skip ROM	$CC	Address a 1-Wire device without its 64-bit ID. This command can only be used if there is a single 1-wire device on the line.
Search ROM	$F0	Reads the 64-bit IDs of all the 1-Wire devices on the line. A process of elimination is used to distinguish each unique device.

The third part, the Memory Function Command, allows the BASIC Stamp to address specific memory locations, or features, of the 1-wire device.

user program; giving the appearance that it is polling "in the background". This feature should not be confused with the concept of interrupts, as the BASIC Stamp *does not support true interrupts*.

A SIMPLE POLLIN EXAMPLE.

The following is an example of the POLLIN command:

```
POLLIN 0, 0
POLLMODE 2
```

The POLLIN command in the above code will cause the BASIC Stamp to set I/O pin 0 to an input mode and get ready to poll it for a low (0) state. The BASIC Stamp will not actually start polling until it is set to the appropriate mode, however. The second line, POLLMODE, initiates the polling process (see the POLLMODE description for more information). From then on, as the BASIC Stamp executes the rest of the program, it will check for a low level (logic 0) on I/O pin 0, in-between instructions.

SETTING ONE OF THE POSSIBLE ACTIONS: POLLED-OUTPUTS

In the code above, no obvious action will be noticed since we didn't tell the BASIC Stamp what to do when it detects a change on the I/O pin. One possible action the BASIC Stamp can be instructed to take is to change the state of an output, called a polled-output. Take a look at the next example:

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

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

In this example, in addition to an endless loop, we've added another polling command called POLLOUT (see the POLLOUT description for more information). Our POLLOUT command tells the BASIC Stamp to set I/O pin 1 to an output mode and set it high (1) when it detects the desired poll state. The poll state is the low (0) level on I/O pin 0 that POLLIN told it to look for. If the polled-input pin is high, it will set polled-output pin 0 to low (0), instead.

Once the program reaches the endless loop, at *Main*, it will continuously print "Looping…" on the PC screen. In between reading the DEBUG command and the GOTO command (and vice versa) it will check polled-input pin 0 and set polled-output pin 1 accordingly. In this case, when I/O pin 0 is set low, the BASIC Stamp will set I/O pin 1 high. When I/O

```
' as the capacitor discharges. Try varying the interval between PWM bursts
' (by changing the PAUSE value) and the number of PWM cycles to see their
' effect.

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


Main:
  PWM 0, 100, 10                        ' PWM at 100/255 duty (~50 ms)
  PAUSE 1000                            ' wait one second
  GOTO Main
  END
```

Demo Program (PWM.bs2)

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

```
' PWM.bs2
' Connect a voltmeter (such as a digital multimeter set to its voltage
' range) to the output of the circuit shown in the figure for the PWM
' command (in the manual).  Run the program and observe the readings on
' the meter. They should come very close to 1.96V, then decrease slightly
' as the capacitor discharges. Try varying the interval between PWM bursts
' (by changing the PAUSE value) and the number of PWM cycles to see their
' effect.

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

#SELECT $STAMP
  #CASE BS2, BS2E
    CycAdj      CON     $100            ' x 1.0, cycle adjustment (for ms)
  #CASE BS2SX
    CycAdj      CON     $280            ' x 2.5
  #CASE BS2P
    CycAdj      CON     $187            ' x 1.53
  #CASE BS2PE
    CycAdj      CON     $09E            ' x 0.62
  #CASE BS2PX
    CycAdj      CON     $280            ' x 2.5
#ENDSELECT

Cycles          CON     50

Main:
  PWM 0, 100, (Cycles */ CycAdj)        ' PWM at 100/255 duty (~50 ms)
  PAUSE 1000                            ' wait one second
  GOTO Main
  END
```

All 2

Demo Program (WRITE.bs2)

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

```
' WRITE.bs2
' This program writes some data to EEPROM and then reads them back out
' and displays the data in the Debug window.

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

idx             VAR     Byte            ' loop control
value           VAR     Word(3)         ' value(s)


Main:
  WRITE 0, 100                          ' single byte
  WRITE 1, Word 1250                    ' single word
  WRITE 3, 45, 90, Word 725             ' multi-value write

Read_EE:
  FOR idx = 0 TO 6                      ' show raw bytes in EE
    READ idx, value
    DEBUG DEC1 idx, " : ", DEC value, CR
  NEXT
  DEBUG CR

  ' read values as stored

  READ 0, value
  DEBUG DEC value, CR
  READ 1, Word value
  DEBUG DEC value, CR
  READ 3, value(0), value(1), Word value(2)
  FOR idx = 0 TO 2
    DEBUG DEC value(idx), CR
  NEXT
  END
```

Table 5.125 lists the XOUT command codes and their functions:

Table 5.125: XOUT Commands and Their Function.

Command	Value	Function
UNITON	%10010	Turn on the currently selected unit.
UNITOFF	%11010	Turn off the currently selected unit.
UNITSONf	%11100	Turn off all modules in this house code.
LIGHTSON	%10100	Turn on all lamp modules in this house code.
DIM	%11110	Reduce brightness of currently selected lamp.
BRIGHT	%10110	Increase brightness of currently selected lamp.

Note: In most applications, it's not necessary to know the code for a given X-10 instruction. Just use the command constant (UnitOn, Dim, etc.) instead. But knowing the codes leads to some interesting possibilities. For example, XORing a UnitOn command with the value %1000 turns it into a UnitOff command, and vice-versa. This makes it possible to write the equivalent of an X-10 "toggle" instruction.

A SIMPLE XOUT EXAMPLE: TURNING AN APPLIANCE ON.

Here is an example of the XOUT instruction:

```
Mpin            PIN     0               ' modulation pin
Zpin            PIN     1               ' zero-cross input

HouseA          CON     0               ' House code A = 0
Unit1           CON     0               ' Unit code 1 = 0

XOUT Mpin, Zpin, [HouseA\Unit1]         ' get Unit1's attention
XOUT Mpin, Zpin, [HouseA\UNITON]        ' turn it on
```

COMBINING MULTIPLE COMMANDS.

You can combine those two XOUT instructions into one like so:

```
XOUT  Mpin, Zpin, [HouseA\Unit1\2, HouseA\UNITON]          ' Unit 1 on.
```

Note that to complete the attention-getting code HouseA\Unit1 we tacked on the normally optional cycles entry \2 to complete the command before beginning the next one. Always specify two cycles in multiple commands unless you're adjusting the brightness of a lamp module.

DIMMING LIGHTS.

Here is an example of a lamp-dimming instruction:

```
Mpin            PIN     0               ' modulation pin
Zpin            PIN     1               ' zero-cross input

HouseA          CON     0               ' House code A = 0
Unit1           CON     0               ' Unit code 1 = 0

XOUT Mpin, Zpin, [HouseA\Unit1]         ' get Unit1's attention
XOUT Mpin, Zpin, [HouseA\UNITOFF\2]     ' turn it off
XOUT Mpin, Zpin, [HouseA\DIM\10]        ' dim half way
```

— F —

— G —

— H —

— I —

— S —