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/pbasic2e-p

(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 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.

THE IDENTIFICATION FUNCTION.

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.

Lets look at the syntax and examples for each conditional compile directive. For an explanation of syntax conventions, see page 128.

#DEFINE SYNTAX.

#DEFINE ***Symbol*** **{ = *Value* }**

#DEFINE allows the programmer to create custom, compile-time, symbols for use within conditional compile control structures.

- ***Symbol*** is a unique symbol name that will optionally represent a *Value*.
- ***Value*** is an optional constant/expression specifying the value of *Symbol*. If the value parameter is omitted, *Symbol* is defined as true (-1).

Example:

```
' {$PBASIC 2.5}

#DEFINE DebugMode

#IF DebugMode #THEN DEBUG "Debugging."
STOP
```

In the example above, the #DEFINE statement defines *DebugMode* to be "true" (-1), since there is no *Value* argument provided. The second line is another conditional compile statement, #IF…#THEN (see below for more information) which evaluates the state of *DebugMode,* determines it is true and then allows the following DEBUG statement to be compiled into the program. The last line, STOP, is compiled into the program afterwards. The result of compiling this example is a program with only two executable statements, DEBUG "Debugging", CR and STOP. The real power of this example, however, is more obvious when you comment out, or remove, the #DEFINE line. Look at the next example, below:

```
' {$PBASIC 2.5}

' #DEFINE DebugMode

#IF DebugMode #THEN DEBUG "Dubugging."
STOP
```

Here we commented out the #DEFINE line, effectively removing that line from the program. This means that the symbol *DebugMode* will be undefined, and undefined conditional compile symbols are treated as

False (0). Upon compiling this example, the #IF…#THEN statement will evaluate *DebugMode*, which is False (because it is undefined) and then will not allow the DEBUG statement to be compiled. Only the STOP command will be compiled into the program in this example. This is a very powerful feature for quickly removing many DEBUG statements (or other statements) from a program when you're done developing it, but leaving the possibility of re-enabling all those statements should further maintenance be required at a later time.

The optional *Value* argument can be used, for example, to select modes of operation:

```
' {$PBASIC 2.5}

#DEFINE SystemMode = 2

#IF SystemMode = 1 #THEN
  HIGH 1
#ELSE
  LOW 1
#ENDIF
```

In the example above, the first line defines *SystemMode* to be equal to 2. The #IF…#THEN statement evaluates the state of *SystemMode*, determines it is 2, so the condition is false, and then it skips the statement after #THEN and allows the statement following #ELSE to be compiled into the program.

Note, conditional compile directives are evaluated just before the program is compiled, so variables and named constants cannot be referenced within a conditional compile definition. Compile-time symbols created with #DEFINE can, however, be referenced by conditional compile commands.

#IF...#THEN SYNTAX.

#IF *Condition(s)* #THEN
  Statement(s)
{ #ELSE
  ***Statement(s)* }**
#ENDIF

```
SYMBOL    value1   =    W0
SYMBOL    value2   =    W1

value1 = 1000
value2 = 19
value1 = value1 * value2              ' Multiply value1 by value2
DEBUG  value1                         ' Show the result (19000)
```

-- or --

```
value1    VAR   Word
value2    VAR   Word
value1 = 1000
value2 = - 19
value1 = value1 * value2              ' Multiply value1 by value2
DEBUG  SDEC  ?  value1                ' Show the result (-19000)
```

MULTIPLY HIGH: **

The Multiply High operator (**) multiplies variables and/or constants, returning the high 16 bits of the result. When you multiply two 16-bit values, the result can be as large as 32 bits. Since the largest variable supported by PBASIC is a word (16 bits), the highest 16 bits of a 32-bit multiplication result are normally lost. The ** (double-star) instruction gives you these upper 16 bits. For example, suppose you multiply 65000 ($FDE8) by itself. The result is 4,225,000,000 or $FBD46240. The * (star, or normal multiplication) instruction would return the lower 16 bits, $6240. The ** instruction returns $FBD4.

```
SYMBOL    value1   =    W0
SYMBOL    value2   =    W1

value1 = $FDE8
value2 = value1 ** value1             ' Multiply $FDE8 by itself
DEBUG  $value2                        ' Return high 16 bits ($FBD4)
```

-- or --

```
value1    VAR   Word
value2    VAR   Word

value1 = $FDE8
value2 = value1 ** value1             ' Multiply $FDE8 by itself
DEBUG  HEX  ?  value2                 ' Return high 16 bits  ($FBD4)
```

An interesting application of the ** operator allows you no multiply a number by a fractional value less than one. The fraction must be expressed in units of 1/65536. To find the fractional ** argument,

BRANCH

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

1 **BRANCH** *Offset*, (*Address0*, *Address1*, …*AddressN*)

All 2 **BRANCH** *Offset*, [*Address0*, *Address1*, …*AddressN*]

Function

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

Go to the address specified by offset (if in range).

- ***Offset*** is a variable/constant/expression (0 – 255) that specifies the index of the address, in the list, to branch to (0 – N).
- ***Addresses*** are labels that specify where to go. BRANCH will ignore any list entries beyond offset 255.

Quick Facts

Table 5.3: BRANCH Quick Facts.

	BS1	All BS2 Models
Limit of *Address* Entries	Limited only by memory	256
Related Commands	None	ON...GOTO

Explanation

The BRANCH instruction is useful when you want to write something like this:

```
IF value = 0 THEN Case_0              ' when value is 0, jump to Case_0
IF value = 1 THEN Case_1              ' when value is 1, jump to Case_1
IF value = 2 THEN Case_2              ' when value is 2, jump to Case_2
```

1 BS1 syntax not shown here.

You can use BRANCH to organize this into a single statement:

```
BRANCH value, [Case_0, Case_1, Case_2]
```

This works exactly the same as the previous IF...THEN example. If the value isn't in range (in this case if *value* is greater than 2), BRANCH does nothing and the program continues with the next instruction after BRANCH.

BRANCH can be teamed with the LOOKDOWN instruction to create a simplified SELECT...CASE statement. See LOOKDOWN for an example.

DATA

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

(See EEPROM)

All 2

{ *Symbol* } **DATA** *DataItem* { , *DataItem…* }

Function

Write data to the EEPROM during program download.

- ***Symbol*** is an optional, unique symbol name that will be automatically defined as a constant equal to the location number of the first data item.
- ***DataItem*** is a constant/expression (0 – 65535) indicating a value, and optionally how to store the value.

Quick Facts

Table 5.7: DATA Quick Facts.

	All BS2 Models
Special Notes	Writes values to EEPROM during download in blocks of 16 bytes. Writes byte or word-sized values. Can be used to decrease program size.
Related Commands	READ and WRITE

Explanation

When you download a program into the BASIC Stamp, it is stored in the EEPROM starting at the highest address (2047) and working towards the lowest address. Most programs don't use the entire EEPROM, so the lower portion is available for other uses. The DATA directive allows you to define a set of data to store in the available EEPROM locations. It is called a "directive" rather than a "command" because it performs an activity at compile-time rather than at run-time (i.e.: the DATA directive is not downloaded to the BASIC Stamp, but the data it contains is downloaded).

WRITING SIMPLE, SEQUENTIAL DATA.

The simplest form of the DATA directive is something like the following:

```
DATA        100, 200, 52, 45
```

This example, when downloaded, will cause the values 100, 200, 52 and 45 to be written to EEPROM locations 0, 1, 2 and 3, respectively. You can then use the READ and WRITE commands in your code to access these locations and the data you've stored there.

displays "00165". Notice that leading zeros? The display is "fixed" to 5 digits, no more and no less. Any unused digits will be filled with zeros.

Using DEC4 in the same code would display "0165". DEC3 would display "165". What would happen if we used DEC2? Regardless of the number, the BASIC Stamp will ensure that it is always the exact number of digits you specified. In this case, it would truncate the "1" and only display "65".

Using the fixed-width version of the formatters in the Signed/Unsigned code above, may result in the following code:

```
x       VAR     Word

x = -65
DEBUG "Signed:   ", SDEC5 x, "  ", ISHEX4 x, "  ", ISBIN16 x, CR
DEBUG "Unsigned: ", DEC5 x, "  ", IHEX4 x, "  ", IBIN16 x
```

and displays:

```
Signed:    -00065    -$0041    -%0000000001000001
Unsigned:  65471    $FFBF    %1111111110111111
```

Note: The columns don't line up exactly (due to the extra "sign" characters in the first row), but it certainly looks better than the alternative.

DISPLAYING STRINGS (BYTE ARRAYS).

If you have a string of characters to display (a byte array), you can use the STR formatter to do so. The STR formatter has two forms (as shown in Table 5.11) for variable-width and fixed-width data. The example below is the variable-width form.

VARIABLE-WIDTH STRINGS.

```
x       VAR     Byte(5)

x(0) = "A"
x(1) = "B"
x(2) = "C"
x(3) = "D"
x(4) = 0
DEBUG STR x
```

This code displays "ABCD" on the screen. In this form, the STR formatter displays each character contained in the byte array until it finds a character that is equal to 0 (value 0, not "0"). This is convenient for use with the SERIN command's STR formatter, which appends 0's to the end of variable-width character string inputs. NOTE: If your byte array

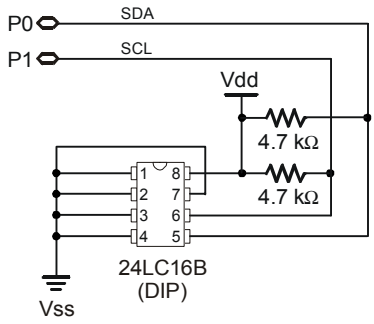

Figure 5.8: Example Circuit for the I2CIN command and a 24LC16B EEPROM.

Note: The 4.7 kΩ resistors are required for the I2CIN command to function properly.

RECEIVING FORMATTED DATA.

The I2CIN command's *InputData* argument is similar to the SERIN command's *InputData* argument. This means data can be received as ASCII character values, decimal, hexadecimal and binary translations and string data as in the examples below. (Assume the 24LC16B EEPROM is used and it has the string, "Value: 3A:101" stored, starting at location 0).

```
value   VAR     Byte(13)

I2CIN 0, $A1, 0, [value]            ' receive the ASCII value for "V"
I2CIN 0, $A1, 0, [DEC value]        ' receive the number 3
I2CIN 0, $A1, 0, [HEX value]        ' receive the number $3A
I2CIN 0, $A1, 0, [BIN value]        ' receive the number %101
I2CIN 0, $A1, 0, [STR value\13]     ' receive the string "Value: 3A:101"
```

Table 5.33 and Table 5.34 below list all the available special formatters and conversion formatters available to the I2CIN command. See the SERIN command for additional information and examples of their use.

Table 5.33: I2CIN Special Formatters.

Special Formatter	Action
SKIP *Length*	Ignore *Length* bytes of characters.
SPSTR L	Input a character stream 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).
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).

I2COUT

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

I2COUT *Pin*, *SlaveID*, { *Address* { *LowAddress* }, } [*OutputData*]

Function

Send data to 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 I2COUT command and both will be set to input mode by the end of the I2COUT 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 send data to. 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.
- ***OutputData*** is a list of variables, constants, expressions and formatters that tells I2COUT how to format outgoing data. I2COUT 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.

POLLRUN

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

POLLRUN *ProgramSlot*

Function

Specify a program to run upon a polled-input event.

- ***ProgramSlot*** is a variable/constant/expression (0 – 7) that specifies the program slot to run when a polled-input event occurs.

Quick Facts

Table 5.80: POLLRUN Quick Facts.

	BS2p, BS2pe, and BS2px
Default *ProgramSlot*	The default polled-run slot is 0. If no POLLRUN command is given and a poll mode of 3 or 4 is set, the program in slot 0 will run in response to a polled-input event.
Special Notes	• If both polled-outputs and polled-run are active, the polled-output event will occur before the polled-run event.
Useful SPRAM locations	Locations 128 – 135 hold polled interrupt status. See Table 5.77 in the POLLMODE command section for more information.
Related commands	POLLMODE, POLLIN, POLLOUT, POLLWAIT and RUN

Explanation

The POLLRUN command is used to specify a program slot to run in response to a polled event. This activity can occur in between any two instructions within the rest of the PBASIC program.

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

PULSOUT

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

1 All 2

PULSOUT *Pin*, *Duration*

Function

1

NOTE: Expressions are not allowed as arguments on the BS1. The range of the *Pin* argument on the BS1 is 0 – 7.

Generate a pulse on *Pin* with a width of *Duration*.

- ***Pin*** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This pin will be set to output mode.
- ***Duration*** is a variable/constant/expression (0 – 65535) that specifies the duration of the pulse. The unit of time for *Duration* is described in Table 5.84.

Quick Facts

Table 5.84: PULSOUT Quick Facts.

	BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px
***Duration* units**	10 µs	2 µs	2 µs	0.8 µs	0.8 µs	2 µs	0.8 µs
Maximum Pulse Width	655.35 ms	131.07 ms	131.07 ms	52.428 ms	52.428 ms	131.07 ms	52.428 ms
Related Command	PULSIN						

Explanation

PULSOUT sets *Pin* to output mode, inverts the state of that pin; waits for the specified *Duration*; then inverts the state of the pin again; returning the bit to its original state. The unit of *Duration* is described in Table 5.84. The following example will generate a 100 µs pulse on I/O pin 7 (of the BS2):

```
PULSOUT 7, 50                          ' generate 100 us pulse on P7
```

CONTROLLING THE POLARITY OF THE PULSE.

The polarity of the pulse depends on the state of the pin before the command executes. In the example above, if pin 7 was low, PULSOUT would produce a positive (high) pulse. If the pin was high, PULSOUT would produce a negative (low) pulse.

WATCH OUT FOR UNDESIRABLE PULSE GLITCHES.

If the pin is an input, the output state bit, OUT7 (PIN7 on the BS1) won't necessarily match the state of the pin. What happens then? For example: pin 7 is an input (DIR7 = 0) and pulled high by a resistor as shown in Figure 5.29a. Suppose that pin 7 is low when we execute the instruction:

```
PULSOUT 7, 5                           ' generate pulse on P7
```

```
' heads and tails thrown.

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

Btn             PIN     7                       ' button input

flip            VAR     Word                    ' a random number
coin            VAR     flip.BIT0               ' Bit0 of the random number
trials          VAR     Byte                    ' number of flips
heads           VAR     Byte                    ' throws that come up heads
tails           VAR     Byte                    ' throws that come up tails
btnWrk          VAR     Byte                    ' workspace for BUTTON


Start:
  DEBUG CLS, "Press button to start"

Main:
  FOR trials = 1 TO 100                         ' flip coin 100 times

Hold:
    RANDOM flip                                 ' randomize while waiting
    BUTTON Btin, 0, 250, 100, btnWrk, 0, Hold   ' wait for button press
    IF (coin = 0) THEN                          ' 0 = heads, 1 = tails
      DEBUG CR, "Heads!"
      heads = heads + 1                         ' increment heads counter
    ELSE
      DEBUG CR, "Tails..."
      tails = tails + 1                         ' increment tails counter
    ENDIF
  NEXT

Done:
  DEBUG CR, CR, "Heads: ", DEC heads, " Tails: ", DEC tails
  END
```

percent of the total change in voltage that they will undergo. More importantly, the value τ is used in the generalized RC timing calculation. Tau's formula is just R multiplied by C:

$\tau = R \times C$

CALCULATING CHARGE AND DISCHARGE TIME.

The general RC timing formula uses τ to tell us the time required for an RC circuit to change from one voltage to another:

$time = -\tau * (\ln (V_{final} / V_{initial}))$

In this formula ln is the natural logarithm; it's a key on most scientific calculators. Let's do some math. Assume we're interested in a 10 k resistor and 0.1 µF cap. Calculate τ:

$\tau = (10 \times 10^{3}) \times (0.1 \times 10^{-6}) = 1 \times 10^{-3}$

The RC time constant is 1×10^{-3} or 1 millisecond. Now calculate the time required for this RC circuit to go from 5V to 1.4V (as in Figure 5.33a):

$time = -1 \times 10^{-3} \times (\ln(1.4v / 5.0v)) = 1.273 \times 10^{-3}$

THE RC TIME EQUATION.

On the BS2, the unit of time is 2µs (See Table 5.87), that time (1.273×10^{-3}) works out to 636 units. With a 10 kΩ resistor and 0.1 µF cap, RCTIME would return a value of approximately 635. Since $V_{initial}$ and V_{final} doesn't change, we can use a simplified rule of thumb to ***estimate*** RCTIME results for circuits like Figure 5.33a:

RCTIME units = 635 x R (in kΩ) x C (in µF)

DETERMINING HOW LONG TO CHARGE OR DISCHARGE THE CAPACITOR BEFORE EXECUTING RCTIME.

Another handy rule of thumb can help you calculate how long to charge/discharge the capacitor before RCTIME. In the example above that's the purpose of the HIGH and PAUSE commands. A given RC charges or discharges 98 percent of the way in 5 time constants (5 x R x C). In Figure 5.33, the charge/discharge current passes through the 220 Ω series resistor and the capacitor. So if the capacitor were 0.1 µF, the minimum charge/discharge time should be:

Charge time = $5 \times 220 \times (0.1 \times 10^{-6}) = 110 \times 10^{-6}$

```
  FOR idx = 0 TO 4                      ' read/display table values
    READ (slotNum * 5) + idx, value
    DEBUG DEC3 value, " "
  NEXT
  DEBUG CR
  PAUSE 1000

  RUN 1                                 ' run Slot 1 pgm
```

Demo Program (RUN2.bsx)

NOTE: This example program was written for the BS2sx but can be used with the BS2e, BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

```
' RUN2.bsx
' This example demonstrates the use of the RUN command.  First, the SPRAM
' location that holds the current slot is read using the GET command to
' display the currently running program number.  Then a set of values
' (based on the program number) are displayed on the screen.  Afterwards,
' program number 0 is run. This program is a BS2sx project consisting of
' RUN1.BSX and RUN2.BSX, but will run on all multi-slot BASIC Stamp models.

' {$STAMP BS2sx}
' {$PBASIC 2.5}

#SELECT $STAMP                          ' set SPRAM of slot number
  #CASE BS2
    #ERROR "Multi-slot BASIC Stamp required."
  #CASE BS2E, BS2SX
    Slot        CON     63
  #CASE BS2P, BS2PE, BS2PX
    Slot        CON     127
#ENDSELECT

slotNum         VAR     Nib             ' current slot
idx             VAR     Nib             ' loop counter
value           VAR     Byte            ' value from EEPROM

EEtable         DATA    100, 40, 80, 32, 90
                DATA    200, 65, 23, 77, 91


Setup:
  GET Slot, slotNum                     ' read current slot
  DEBUG  "Program #", DEC slotNum, CR   ' display

Main:
  FOR idx = 0 TO 4                      ' read/display table values
    READ (slotNum * 5) + idx, value
    DEBUG DEC3 value, " "
  NEXT
  DEBUG CR
  PAUSE 1000

  RUN 0                                 ' back to Slot 0 pgm
```

Value1 TO *Value2*

This indicates a range of Value1 to Value2, inclusive. For example, *20* TO *23* means 20, 21, 22 and 23. Similarly, *"A"* TO *"F"* means all the characters in the range "A" through "F".

Finally, multiple conditions can be included in a single CASE by separating them with commas (,). For example,

```
CASE  20, 25 TO 30, >100
```

will evaluate to True if the *Expression* (from the SELECT statement) is equal to 20, or is in the range 25 through 30, or is greater than 100.

An example will help clarify this function.

```
'{$PBASIC 2.5}

guess     VAR   WORD

DEBUG "Guess my favorite number: "         ' prompt user

DO
  DEBUGIN DEC Guess                        ' get answer

  SELECT guess
    CASE < 100                             ' less than 100?
      DEBUG CR, "Not even close. Higher."
    CASE > 140                             ' greater than 140?
      DEBUG CR, "Too high."
    CASE 100 TO 120, 126 TO 140            ' 100-120 or 126-140?
      DEBUG CR, "Getting closer..."
    CASE 123                               ' 123? Got it!
      DEBUG CR, "That's it!  123!"
      DEBUG CR, "Good Guessing!"
      STOP
    CASE 121 TO 125                        ' close to 123?
      DEBUG CR, "You're so close!"
  ENDSELECT

  DEBUG CR, "Try again: "                  ' encourage another try
LOOP
```

This program will ask the user to guess a number, store that value in *guess* and check the results in the SELECT statement. If *guess* is less than 100, the first CASE is true and BASIC Stamp will display "Not even close.

USING SERIN TO WAIT FOR SPECIFIC DATA BEFORE PROCESSING.

The SERIN command can also be configured to wait for specified data before it retrieves any additional input. For example, suppose a device that is attached to the BASIC Stamp is known to send many different sequences of data, but the only data you desire happens to appear right after the unique characters, "XYZ". The BS1 has optional *Qualifier* arguments for this purpose. On all BS2 models, a special formatter called WAIT can be used for this.

```
SYMBOL  serData = B2

SERIN 1, N2400, ("XYZ"), #serData
```

-- or --

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

```
serData VAR     Byte

SERIN 1, 16780, [WAIT("XYZ"), DEC serData]
```

The above code waits for the characters "X", "Y" and "Z" to be received, in that order, and then it looks for a decimal number to follow. If the device in this example were to send the characters "XYZ100" followed by a carriage return or some other non-decimal numeric character, the *serData* variable would end up with the number 100 after the SERIN line finishes. If the device sent some data other than "XYZ" followed by a number, the BASIC Stamp would continue to wait at the SERIN command.

The BS1 will accept an unlimited number of *Qualifiers*. All BS2 models will only accept up to six bytes (characters) in the WAIT formatter.

USING ASCII CODES AND CASE SENSITIVITY.

Keep in mind that when we type "XYZ" into the SERIN command, the BASIC Stamp actually uses the ASCII codes for each of those characters for its tasks. We could also have typed: 88, 89, 90 in place of "XYZ" and the code 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. Also note, serial communication with the BASIC Stamp is case sensitive. If the device mentioned above sent, "xYZ" or "xyZ", or some other combination of lower and upper-case characters, the BASIC Stamp would have ignored it because we told it to look for "XYZ" (all capital letters).

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

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]
```

   a. Unmatched settings on the sender and receiver side will cause garbled data transfers or no data transfers. If the data you receive is unreadable, it is most likely a baud rate setting error.
5. If data transmitted to the Stamp Editor's Debug Terminal is garbled, verify the output format.
   a. A common mistake is to send data with SEROUT in ASCII format. For example, SEROUT 16, 84, [0] instead of SEROUT 16, 84, [DEC 0]. The first example will send a byte equal to 0 to the PC, resulting in the Debug Terminal clearing the screen (since 0 is the control character for a clear-screen action).

Demo Program (SEROUT.bs1)

```
' SEROUT.bs1
' This program transmits the string "ABCD" followed by a number and a
' carriage-return at 2400 baud, inverted, N81 format.

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

SYMBOL  SOut            = 1
SYMBOL  Baud            = N2400

SYMBOL  value           = W1


Setup:
  value = 1

Main:
  SEROUT SOut, Baud, ("ABCD", #value)
  value = value + 1
  PAUSE 250
  GOTO Main
  END
```

Demo Program (SERIN_SEROUT1.bs2)

NOTE: This example program was written for BS2's but it can be used with the BS2e, BS2sx, BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

```
' SERIN_SEROUT1.bs2
' Using two BS2-IC's, connect the circuit shown in the SERIN command
' description and run this program on the BASIC Stamp designated as the
' Sender. This program demonstrates the use of Flow Control (FPin).
' Without flow control, the sender would transmit the whole word "Hello!"
' in about 1.5 ms. The receiver would catch the first byte at most; by the
' time it got back from the first 1-second PAUSE, the rest of the data
```

SHIFTOUT – BASIC Stamp Command Reference

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
```