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"

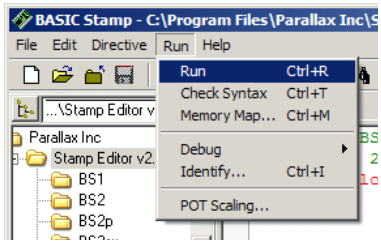
Details	
Product Status	Active
Core Processor	-
Core Size	-
Speed	-
Connectivity	-
Peripherals	-
Number of I/O	-
Program Memory Size	-
Program Memory Type	-
EEPROM Size	-
RAM Size	-
Voltage - Supply (Vcc/Vdd)	-
Data Converters	-
Oscillator Type	-
Operating Temperature	-
Mounting Type	-
Package / Case	-
Supplier Device Package	-
Purchase URL	https://www.e-xfl.com/product-detail/parallax/pbasic2ci-ss

6) Type the line DEBUG "Hello World!" below the compiler directives:

```
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Hello World!"
```

7) Download this program into the BASIC Stamp. You may select Run → Run from the menu bar, press CTRL-R from the keyboard, or click on the Run ► icon on the toolbar.

Figure 2.5: To run your program, you may use the task bar menu or the Run icon.

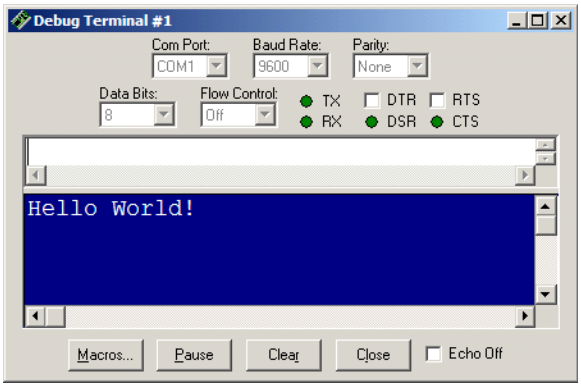


Selecting Run → Run

Using the Run toolbar icon

a) If the program is typed correctly, a progress bar window should appear (perhaps very briefly) showing the download progress. Then a Debug Terminal window should appear and display "Hello World!"

Figure 2.6: Debug Terminal displaying program output



b) If there is a syntax error in the program, the editor will highlight the text in question and display an error message. Review the error, fix the code and then try downloading again.

c) If the error reported a connection problem with the BASIC Stamp, make sure the first line of code indicates the proper module name and verify the programming cable connections, module orientation (in the socket) and that it is properly powered, then try downloading again.

8) Congratulations! You've just written and downloaded your first BASIC Stamp program! The "Hello World!" text that appeared on the screen was sent from the BASIC Stamp, back up the programming cable, to the PC.

```
myBytes         VAR     Byte(10)        ' Define 10-byte array
idx             VAR     Nib             ' Define 4-bit var

FOR idx = 0 TO 9                        ' Repeat with idx = 0, 1, 2...9
  myBytes(idx) = idx * 13               ' Write idx * 13 to each cell
NEXT

FOR idx = 0 TO 9                        ' Repeat with idx = 0, 1, 2...9
  DEBUG ? myBytes(idx)                  ' Show contents of each cell
NEXT
STOP
```

If you run this program, DEBUG will display each of the 10 values stored in the elements of the array: myBytes(0) = 0*13 = 0, myBytes(1) = 1*13 = 13, myBytes(2) = 2*13 = 26 ... myBytes(9) = 9*13 = 117.

A word of caution about arrays: If you're familiar with other BASICs and have used their arrays, you have probably run into the "subscript out of range" error. Subscript is another term for the index value. It is out-of-range when it exceeds the maximum value for the size of the array. For instance, in the example above, myBytes is a 10-cell array. Allowable index numbers are 0 through 9. If your program exceeds this range, PBASIC will not respond with an error message. Instead, it will access the next RAM location past the end of the array. If you are not careful about this, it can cause all sorts of bugs.

If accessing an out-of-range location is bad, why does PBASIC allow it? Unlike a desktop computer, the BASIC Stamp doesn't always have a display device connected to it for displaying error messages. So it just continues the best way it knows how. It's up to the programmer (you!) to prevent bugs. Clever programmers, can take advantage of this feature, however, to perform tricky effects.

Another unique property of PBASIC arrays is this: You can refer to the 0th cell of the array by using just the array's name without an index value. For example:

```
myBytes         VAR     Byte(10)        ' Define 10-byte array

myBytes(0) = 17                         ' Store 17 to 0th cell
DEBUG ? myBytes(0)                      ' Display contents of 0th cell
DEBUG ? myBytes                         ' Also displays 0th cell
```

Demo Program (BRANCH.bs1)


```
' BRANCH.bs1
' This program shows how the value of idx controls the destination of the
' BRANCH instruction.

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

SYMBOL  idx             = B2


Main:
  DEBUG "idx: ", #idx, " "
  BRANCH idx, (Task_0, Task_1, Task_2)          ' branch to task
  DEBUG "BRANCH target error...", CR, CR        ' ... unless out of range

Next_Task:
  idx = idx + 1 // 4                            ' force idx to be 0..3
  GOTO Main

Task_0:
  DEBUG "BRANCHed to Task_0", CR
  GOTO Next_Task

Task_1:
  DEBUG "BRANCHed to Task_1", CR
  GOTO Next_Task

Task_2:
  DEBUG "BRANCHed to Task_2", CR
  GOTO Next_Task
```

Demo Program (BRANCH.bs2)

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

```
' BRANCH.bs2
' This program shows how the value of idx controls the destination of the
' BRANCH instruction.

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

idx             VAR     Nib


Main:
  DEBUG "idx: ", DEC1 idx, " "
  BRANCH idx, [Task_0, Task_1, Task_2]          ' branch to task
  DEBUG "BRANCH target error...", CR, CR        ' ... unless out of range

Next_Task:
  idx = idx + 1 // 4                            ' force idx to be 0..3
```

CONFIGPIN

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

CONFIGPIN *Mode, PinMask*

Function

Configure special properties of I/O pins.

- ***Mode*** is a variable/constant/expression (0 – 3), or one of four predefined symbols, that specifies the I/O pin property to configure: Schmitt Trigger, Logic Threshold, Pull-up Resistor or Output Direction. See Table 5.5 for an explanation of *Mode* values.
- ***PinMask*** is a variable/constant/expression (1 – 65535) that indicates how *Mode* is applied to I/O pins. Each bit of *PinMask* corresponds to an individual I/O pin. A high bit (1) enables the *Mode* and a low bit (0) disables the *Mode* on the corresponding I/O pin.

Quick Facts

Table 5.5: CONFIGPIN Quick Facts.

	BS2px
***Mode* Values**	0 (or SCHMITT): Schmitt Trigger
	1 (or THRESHOLD): Logic Threshold
	2 (or PULLUP): Pull-up Resistor
	3 (or DIRECTION): Output Direction
Related Commands (For DIRECTION Mode)	INPUT and OUTPUT, and the DIRx = # assignment statement

Explanation

The CONFIGPIN command enables or disables special I/O pin properties on all 16 I/O pins at once. There are four properties, or modes, available: Schmitt Trigger, Logic Threshold, Pull-up Resistor, and Output Direction. Each I/O pin on the BS2px contains special hardware dedicated to each of these properties.

OUTPUT DIRECTION.

By default, all BASIC Stamp I/O pins are set to inputs. Enabling the Output Direction mode sets an I/O pin's direction to output. Disabling the Output Direction mode sets an I/O pin's direction to input. This has the same effect as using the OUTPUT or INPUT commands, or the DIRx = # assignment statement to configure I/O pin directions. The following is an example of the CONFIGPIN command using the Output Direction mode:

```
CONFIGPIN  DIRECTION, %0000000100010011
```

THE DATA POINTER (COUNTER).

DATA uses a counter, called a pointer, to keep track of available EEPROM addresses. The value of the pointer is initially 0. When a program is downloaded, the DATA directive stores the first byte value at the current pointer address, then increments (adds 1 to) the pointer. If the program contains more than one DATA directive, subsequent DATAs start with the pointer value left by the previous DATA. For example, if the program contains:

```
DATA        72, 69, 76, 76, 79
DATA        104, 101, 108, 108, 111
```

The first DATA directive will start at location 0 and increment the pointer for each data value it stores (1, 2, 3, 4 and 5). The second DATA directive will start with the pointer value of 5 and work upward from there. As a result, the first 10 bytes of EEPROM will look like the following:

Table 5.8: Example EEPROM Storage.

	EEPROM Location (address)									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Contents	72	69	76	76	79	104	101	108	108	111

WRITING DATA TO OTHER LOCATIONS.

What if you don't want to store values starting at location 0? Fortunately, the DATA directive has an option to specify the next location to use. You can specify the next location number (to set the pointer to) by inserting a *DataItem* in the form @x ;where x is the location number. The following code writes the same data in Table 5.8 to locations 100 through 109:

```
DATA        @100, 72, 69, 76, 76, 79, 104, 101, 108, 108, 111
```

In this example, the first *DataItem* is @100. This tells the DATA directive to store the following *DataItem(s)* starting at location 100. All the *DataItems* to the right of the @100 are stored in their respective locations (100, 101, 102… 109).

In addition, the DATA directive allows you to specify new starting locations at any time within the *DataItem* list. If, for example, you wanted to store 56 at location 100 and 47 at location 150 (while leaving every other location intact), you could type the following:

```
DATA        @100, 56, @150, 47
```

AUTOMATIC CONSTANTS FOR DEFINED DATA.

If you have multiple DATA directives in your program, it may be difficult to remember exactly what locations contain the desired data. For this reason, the DATA directive can optionally be prefixed with a unique

symbol name. This symbol becomes a constant that is set equal to the location number of the first byte of data within the directive. For example,

```
MyNumbers     DATA    @100, 72, 73
```

This would store the values 72 and 73 starting with location 100 and will create a constant, called *MyNumbers*, which is set equal to 100. Your program can then use the *MyNumbers* constant as a reference to the start of the data within a READ or WRITE command. Each DATA directive can have a unique symbol preceding it, allowing you to reference the data defined at different locations.

RESERVING EEPROM LOCATIONS.

There may be a time when you wish to reserve a section of EEPROM for use by your BASIC code, but not necessarily store data there to begin with. To do this, simply specify a *DataItem* within parentheses, as in:

```
DATA          @100, (20)
```

The above DATA directive will reserve 20 bytes of EEPROM, starting with location 100. It doesn't store any values there, rather it simply leaves the data as it is and increments DATA's location pointer by 20. A good reason to do this is when you have a program already downloaded into the BASIC Stamp that has created or manipulated some data in EEPROM. To protect that section of EEPROM from being overwritten by your next program (perhaps a new version of the same program) you can reserve the space as shown above. The EEPROM's contents from locations 100 to 119 will remain intact. NOTE: This only "reserves" the space for the program you are currently downloading; the BASIC Stamp does not know to "reserve" the space for future programs. In other words, make sure use this feature of the DATA directive in every program you download if you don't want to risk overwriting valuable EEPROM data.

IMPORTANT CONCEPT: HOW DATA AND PROGRAMS ARE DOWNLOADED INTO EEPROM.

It is important to realize that EEPROM is not overwritten during programming unless it is needed for program storage, or is filled by a DATA directive specifying data to be written. **During downloading, EEPROM is always written in 16-byte sections if, and only if, any location within that section needs writing.**

WRITING A BLOCK OF THE SAME VALUE.

DATA can also store the same number in a block of consecutive locations. This is similar to reserving a block of EEPROM, above, but with a value added before the first parenthesis.

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.

Note that the WHILE test (loop runs WHILE *Condition* is True) and UNTIL test (loop runs UNTIL *Condition* is True) can be interchanged, but they are generally used as illustrated above.

All 2

Demo Program (DO-LOOP.bs2)

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

```
' DO-LOOP.bs2
' This program creates a little guessing game.  It starts by creating
' a (pseudo) random number between 1 and 10.  The inner loop will run
' until the answer is guessed or 10 tries have been attempted.  The
' outer loop has no condition and will cause the inner loop code to
' run until the BASIC Stamp is reprogrammed.

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

rVal            VAR     Word            ' random value
answer          VAR     Byte            ' game answer
guess           VAR     Byte            ' player guess
tries           VAR     Nib             ' number of tries


Main:
  DO
    RANDOM rVal
    answer = rVal.LOWBYTE */ 10 + 1     ' create 1 - 10 answer
    tries = 0

    DO                                  ' get answer until out of tries
      DEBUG CLS,
            "Guess a number (1 - 10): "
      DEBUGIN DEC guess                 ' get new guess
      tries = tries + 1                 ' update tries count
    LOOP UNTIL ((tries = 10) OR (guess = answer))

    IF (guess = answer) THEN            ' test reason for loop end
      DEBUG CR, "You got it!"
    ELSE
      DEBUG CR, "Sorry ... the answer was ", DEC answer, "."
    ENDIF
    PAUSE 1000
  LOOP                                  ' run again
  END
```

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 5.34: I2CIN 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.

THE I²C PROTOCOL FORMAT.

The I²C protocol has a well-defined standard for the information passed at the start of each transmission. First of all, any information sent must be transmitted in units of 1 byte (8-bits). The first byte, we call the *SlaveID*, is an 8-bit pattern whose upper 7-bits contain the unique ID of the device you wish to communicate with. The lowest bit indicates whether this is a write operation (0) or a read operation (1). Figure 5.9 shows this format.

Figure 5.9: Slave ID Format.

7	6	5	4	3	2	1	0
A_6	A_5	A_4	A_3	A_2	A_1	A_0	R/W

The second byte, immediately following the SlaveID, is the optional *Address*. It indicates the 8-bit address (within the device) containing the data you would like to receive. Note that the *Address* argument is optional and may be left unspecified for devices that don't require an *Address* argument.

USING LONG ADDRESSES.

Some devices require more than 8 bits of address. For this case, the optional *LowAddress* argument can be used for the low-byte of the required address. When using the *LowAddress* argument, the *Address* argument is effectively the high-byte of the address value. For example, if the entire address value is 2050, use 8 for the *Address* argument and 2 for the *LowAddress* argument (8 * 256 + 2 = 2050).

Following the last address byte is the first byte of data. This data byte may be transmitted or received by the BASIC Stamp. In the case of the I2CIN command, this data byte is transmitted by the device and received by the BASIC Stamp. Additionally, multiple data bytes can follow the address, depending on the I²C device. Note that every device has different limitations regarding how may contiguous bytes they can receive or transmit in one session. Be aware of these device limitations and program accordingly.

START AND STOP CONDITIONS AND ACKNOWLEDGMENTS.

Every I²C transmission session begins with a Start Condition and ends with a Stop Condition. Additionally, immediately after every byte is transmitted, an extra clock cycle is used to send or receive an acknowledgment signal (ACK). All of these operations are automatically taken care of by the I2CIN command so that you need not be concerned with them. The general I²C transmission format is shown in Figure 5.10.

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
```

OWIN

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

OWIN ***Pin, Mode,*** **[** ***InputData*** **]**

Function

Receive data from 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 OWIN command and will be set to input mode during the OWIN command and will be set to input mode by the end of the OWIN 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.
- ***InputData*** is a list of variables and modifiers that tells OWIN what to do with incoming data. OWIN 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.

Quick Facts

Table 5.63: OWIN Quick Facts.

	BS2p, BS2pe, and BS2px
Receive 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 Commands	OWOUT

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 OWIN command allows the BASIC Stamp to receive data from a 1-wire device.

A SIMPLE OWIN EXAMPLE.

The following is an example of the OWIN command:

```
result  VAR     Byte

OWIN 0, 1, [result]
```

RANDOM

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

1 All 2

RANDOM ***Variable***

Function

Generate a pseudo-random number.

- ***Variable*** is a variable (usually a word) whose bits will be scrambled to produce a random number. *Variable* acts as RANDOM's input and its result output. Each pass through RANDOM stores the next number, in the pseudorandom sequence, in *Variable*.

Explanation

RANDOM generates pseudo-random numbers ranging from 0 to 65535. They're called "pseudo-random" because they appear random, but are generated by a logic operation that uses the initial value in *Variable* to "tap" into a sequence of 65535 essentially random numbers. If the same initial value, called the "seed", is always used, then the same sequence of numbers is generated. The following example demonstrates this:

1

```
SYMBOL  result           = W1

Main:
  result = 11000
  RANDOM result
  DEBUG result
  GOTO Main
```

-- or --

All 2

```
result          VAR     Word

Main:
  result = 11000
  RANDOM result
  DEBUG DEC ? result
  GOTO Main
```

In this example, the same number would appear on the screen over and over again. This is because the same seed value was used each time; specifically, the first line of the loop sets *result* to 11,000. The RANDOM command really needs a different seed value each time. Moving the "result =" line out of the loop will solve this problem, as in:

```
SYMBOL  result           = W1

Setup:
  result = 11000

Main:
  RANDOM result
  DEBUG result
  GOTO Main
```

-- or --

```
result          VAR     Word

Setup:
  result = 11000

Main:
  RANDOM result
  DEBUG DEC ? result
  GOTO Main
```

Here, *result* is only initialized once, before the loop. Each time through the loop, the previous value of *result*, generated by RANDOM, is used as the next seed value. This generates a more desirable set of pseudorandom numbers.

In applications requiring more apparent randomness, it's necessary to "seed" RANDOM with a more random value every time. For instance, in the demo program below, RANDOM is executed continuously (using the previous resulting number as the next seed value) while the program waits for the user to press a button. Since the user can't control the timing of button presses very accurately, the results approach true randomness.

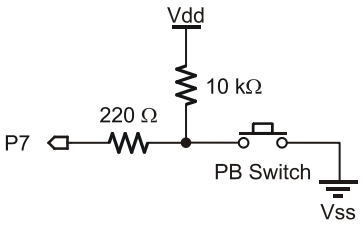


Figure 5.32: RANDOM Button Circuit.

```
Main:
  PAUSE 250                             ' 1/4th second pause
  REVERSE 0                             ' reverse pin 0 I/O direction
  GOTO Main                             ' do forever
```

RUN – BASIC Stamp Command Reference

Explanation

SELECT…CASE is an advanced decision-making structure that is often used to replace compound IF…THEN…ELSE structures. SELECT…CASE statements are good for performing one of many possible actions depending on the value of a single expression.

Upon reaching a SELECT…CASE statement, the BASIC Stamp will evaluate *Expression* once and then compare it to the *Condition(s)* of each CASE until it finds a "case" that evaluates to True, or it runs out of cases to compare to. As soon as a True case is found, the BASIC Stamp executes that CASE's *Statement(s)* and then continues execution on the program line following ENDSELECT.

To understand how SELECT…CASE statements work, it helps to review how IF…THEN statements behave. The condition argument of IF…THEN takes the form:

Value1 Comparison Value2

and *Value1* is "compared" to *Value2* using the *Comparison* operator.

In SELECT...CASE statements, the *Value1* component is always *Expression* and so the format of *Condition(s)* is simplified to:

{ *Comparison* } *Value*

Comparison is optional and can be any of the comparison operators shown in Table 5.93. If C*omparison* is not specified, it is an implied Equal (=) operator. *Value* can be a variable, constant or expression.

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

Table 5.93: Comparison Operators for SELECT...CASE.

Condition(s) also has a special, additional format that can be used to indicate a range of sequential values:

```
  #CASE BS2SX, BS2P
    T1200       CON     2063
    T2400       CON     1021
    T9600       CON     240
    T19K2       CON     110
    T38K4       CON     45
  #CASE BS2PX
    T1200       CON     3313
    T2400       CON     1646
    T9600       CON     396
    T19K2       CON     188
    T38K4       CON     84
#ENDSELECT

Inverted        CON     $4000
Open            CON     $8000
Baud            CON     T38K4 + Inverted

letter          VAR     Byte


Main:
  DO
    SERIN SI\FC, Baud, [letter]         ' receive one byte
    DEBUG letter                        ' display on screen
    PAUSE 1000                          ' wait one second
  LOOP                                  ' repeat forever
  END
```

A SIMPLE SHIFTIN EXAMPLE.

Here is a simple example:

```
result          VAR     Byte

SHIFTIN 0, 1, MSBPRE, [result]
```

Here, the SHIFTIN command will read I/O pin 0 (*Dpin*) and will generate a clock signal on I/O 1 (*Cpin*). The data that arrives on *Dpin* depends on the device connected to it. Let's say, for example, that a shift register is connected and has a value of $AF (10101111) waiting to be sent. Additionally, let's assume that the shift register sends out the most significant bit first, and the first bit is on *Dpin* before the first clock pulse (MSBPRE). The SHIFTIN command above will generate eight clock pulses and sample the data pin (*Dpin*) eight times. Afterward, the *result* variable will contain the value $AF.

CONTROLLING THE NUMBER OF BITS RECEIVED.

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

```
result          VAR     Byte

SHIFTIN 0, 1, MSBPRE, [result\4]
```

Will only input the first 4 bits. In the example discussed above, the *result* variable will be left with %1010.

Some devices return more than 16 bits. For example, most 8-bit shift registers can be daisy-chained together to form any multiple of 8 bits; 16, 24, 32, 40... To solve this, you can use a single SHIFTIN instruction with multiple variables. Each variable can be assigned a particular number of bits with the *Bits* argument. As in:

```
resultLo        VAR     Word
resultHi        VAR     Nib

SHIFTIN 0, 1, MSBPRE, [resultHi\4, resultLo\16]
```

The above code will first shift in four bits into *resultHi* and then 16 bits into *resultLo*. The two variables together make up a 20 bit value.