Welcome to E-XFL.COM

Understanding Embedded - Microcontroller, Microprocessor, FPGA Modules

Embedded - Microcontroller, Microprocessor, and FPGA Modules are fundamental components in modern electronic systems, offering a wide range of functionalities and capabilities. Microcontrollers are compact integrated circuits designed to execute specific control tasks within an embedded system. They typically include a processor, memory, and input/output peripherals on a single chip. Microprocessors, on the other hand, are more powerful processing units used in complex computing tasks, often requiring external memory and peripherals. FPGAs (Field Programmable Gate Arrays) are highly flexible devices that can be configured by the user to perform specific logic functions, making them invaluable in applications requiring customization and adaptability.

Applications of Embedded - Microcontroller,

Details	
Product Status	Active
Module/Board Type	MCU Core
Core Processor	SX48AC
Co-Processor	-
Speed	32MHz
Flash Size	16KB EEPROM
RAM Size	38B
Connector Type	-
Size / Dimension	1.2" x 0.6" (30mm x 15mm)
Operating Temperature	0°C ~ 70°C
Purchase URL	https://www.e-xfl.com/product-detail/parallax/bs2px24

Basic Stamp 2px

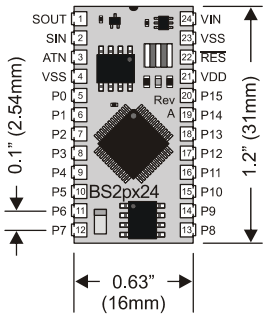

Figure 1.12: BASIC Stamp 2px (Rev. A) (Stock# BS2px-IC)

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

Table 1.7: BASIC Stamp 2px Pin Descriptions.

Pin	Name	Description
1	SOUT	Serial Out: connects to PC serial port RX pin (DB9 pin 2 / DB25 pin 3) for programming.
2	SIN	Serial In: connects to PC serial port TX pin (DB9 pin 3 / DB25 pin 2) for programming.
3	ATN	Attention: connects to PC serial port DTR pin (DB9 pin 4 / DB25 pin 20) for programming.
4	VSS	System ground: (same as pin 23), connects to PC serial port GND pin (DB9 pin 5 / DB25 pin 7) for programming.
5-20	P0-P15	General-purpose I/O pins: each can source and sink 30 mA. However, the total of all pins should not exceed 75 mA (source or sink) if using the internal 5-volt regulator. The total per 8-pin groups P0 – P7 or P8 – 15 should not exceed 100 mA (source or sink) if using an external 5-volt regulator.
21	VDD	5-volt DC input/output: if an unregulated voltage is applied to the VIN pin, then this pin will output 5 volts. If no voltage is applied to the VIN pin, then a regulated voltage between 4.5V and 5.5V should be applied to this pin.
22	RES	Reset input/output: goes low when power supply is less than approximately 4.2 volts, causing the BASIC Stamp to reset. Can be driven low to force a reset. This pin is internally pulled high and may be left disconnected if not needed. Do not drive high.
23	VSS	System ground: (same as pin 4) connects to power supply's ground (GND) terminal.
24	VIN	Unregulated power in: accepts 5.5 - 12 VDC (7.5 recommended), which is then internally regulated to 5 volts. Must be left unconnected if 5 volts is applied to the VDD (+5V) pin.

editor will have its own tab at the top of the page labeled with the name of the file, as seen in Figure 3.2. The full file path of the currently displayed source code appears in the title bar. Source code that has never been saved to disk will default to "Untitled#"; where # is an automatically generated number. A user can switch between source code files by simply pointing and clicking on a file's tab or by pressing Ctrl+Tab or Ctrl+Shift+Tab while the main edit pane is active.

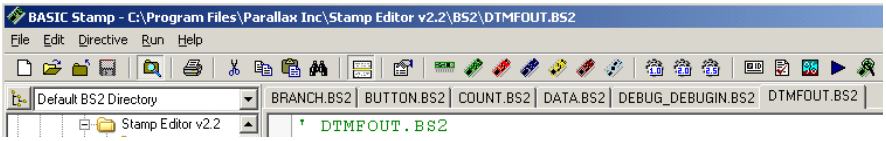

Figure 3.2: Example Editor Tabs. Shown with 6 separate files open; Title Bar shows current code's file path.

The status of the active source code is indicated in the status bar below the main edit pane and integrated explorer panel. The status bar contains information such as cursor position, file save status, download status and syntax error/download messages. The example in Figure 3.3 indicates that the source code tokenized successfully.

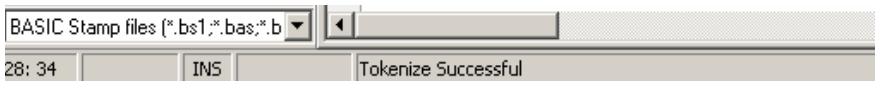

Figure 3.3: Status Bar beneath the Main Edit Pane.

SPLIT WINDOW VIEW.

Each editor pane can be individually split into two views of the same source code. This can be done via the Split button on the toolbar, pressing Ctrl-L, or clicking and dragging the top or bottom border of the editor pane with the mouse.

Once split, the top and bottom edit controls allow viewing of different areas of the same source code; this can be handy when needing to keep variable declarations or a particular routine in view while modifying a related section of code elsewhere. Note that the Split button and Ctrl+L shortcut act like a toggle function, splitting or un-splitting the edit pane.

THE EEPROM MAP.

The EEPROM map is shown in two scales. The main view is the detailed EEPROM map, which displays the data in hexadecimal format in each location. The condensed EEPROM map is the vertical region on the left that shows a small-scale view of the entire EEPROM; the red square over it corresponds to the scroll bar handle in the detailed EEPROM map and indicates the portion of the EEPROM that is currently visible in the detailed EEPROM map.

Checking the Display ASCII checkbox switches the detailed EEPROM display from hexadecimal to ASCII. In this program, the textual data can be read right off the EEPROM map when using this option.

Two important points to remember about this map are: 1) it only indicates how your program will be downloaded to the BASIC Stamp module; it does not "read" the BASIC Stamp memory, and 2) for all BS2 models, fixed variables like B3 and W1 and any aliases do not show up on the memory map as memory used. The editor ignores fixed variables when it arranges automatically allocated variables in memory. Remember, fixed and allocated variables can overlap.

THE DEBUG TERMINAL.

The Debug Terminal window provides a convienent display for data received from a BASIC Stamp during run-time, and also allows for the transmission of characters from the PC keyboard to the BASIC Stamp. The Debug Terminal is automatically opened and configured when a PBASIC program, containing a DEBUG command, is downloaded. You can manually open a Debug Terminal one of three ways: select Run → Debug → New, press Ctrl+D on the keyboard, or click on the Debug Terminal toolbar button. Up to four (4) Debug Terminals can be open at once (on four different ports) and all can be left open while editing and downloading source code.

Figure 3.14 below shows the demo program DEBUG_DEBUGIN.bs2 in the edit pane, and the Debug Terminal that opens when this program is run.

Using the BASIC Stamp Editor

```
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
```

AUXIO

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

AUXIO

Function

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

Quick Facts

Table 5.2: AUXIO Quick Facts.

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

Explanation

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

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

For the CONFIGPIN command's SCHMITT mode, a high bit (1) in the *PinMask* argument enables the Schmitt Trigger on the corresponding I/O pin and a low bit (0) disables the Schmitt Trigger. The following example sets Schmitt Triggers on I/O pins 7, 6, 5, and 4, and sets all other I/O pins to normal mode.

```
CONFIGPIN  SCHMITT, %0000000011110000
```

Schmitt Trigger mode can be activated for all pins, regardless of pin direction, but really matters only when the associated pin is set to input mode.

Demo Program (CONFIGPIN.bpx)

NOTE: This example program can be used only with the BS2px.

```
' CONFIGPIN.BPX
' This example demonstrates the use of the CONFIGPIN command.
' All I/O pins are set to inputs with various combinations of
' Pull-Up Resistor, Logic Threshold and Schmitt-Trigger properties.
' While running, this program will constantly display the state of all
' input pins along with an indication of the configuration for each group
' of pins.  Try connecting different input signals to the I/O pins (such as
' buttons, a function generator with a slowing sweeping signal (0 to 5
' VDC)) or simply running your fingers across the I/O pins and note how
' they react based upon their configured property.

' {$STAMP BS2px}
' {$PBASIC 2.5}

#IF $STAMP <> BS2PX #THEN
  #ERROR "This program requires a BS2px."
#ENDIF

Setup:
  CONFIGPIN DIRECTION, %0000000000000000   'Set all I/O pins to inputs
  CONFIGPIN PULLUP,    %1111111111110000   'Enable pull-ups on pins 4 - 15
  CONFIGPIN THRESHOLD, %0000111100000000   'Set P8-P11 to CMOS, others TTL
  CONFIGPIN SCHMITT,   %1111000000000000   'Enable Schmitt-Triggers P12-P15

  DEBUG CLS
  DEBUG "         BS2px INPUT PIN CONFIGURATION TEST", CR,
        "=======================================================", CR,
        "      P15-P12: Pull-Up Resistors, TTL & Schmitt-Triggers", CR,
        "     /", CR,
        "    /    P11-P8: Pull-Up Resistors & CMOS", CR,
        "   /    /", CR,
        "  |    /     P7-P4: Pull-Up Resistors & TTL", CR,
        "  |    |    /", CR,
        "  |    |    |     P3-P0: Normal", CR,
        "  |    |    |    /", CR,
```

DO…LOOP

BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px

All 2

DO { WHILE | UNTIL *Condition(s)* }
** *Statement(s)***
LOOP { WHILE | UNTIL *Condition(s)* }

NOTE: DO...LOOP requires the PBASIC 2.5 compiler directive.

Function

Create a repeating loop that executes the *Statement(s)*, one or more program lines that form a code block, between DO and LOOP, optionally testing *Condition(s)* before or after the *Statement(s)*.

- ***Condition*** is an optional variable/constant/expression (0 - 65535) which determines whether the loop will run or terminate. *Condition* must follow WHILE or UNTIL.
- ***Statement*** is any valid PBASIC instruction.

Quick Facts

Table 5.18: DO...LOOP Quick Facts.

	All BS2 Models
Maximum Nested Loops	16
WHILE *Condition* Evaluation	Run loop if *Condition* evaluates as true
UNTIL *Condition* Evaluation	Terminate loop if *Condition* evaluates as true
Related Commands	FOR...NEXT and EXIT

Explanation

DO...LOOP loops let a program execute a series of instructions indefinitely or until a specified condition terminates the loop. The simplest form is shown here:

```
' {$PBASIC 2.5}

DO
  DEBUG "Error...", CR
  PAUSE 2000
LOOP
```

In this example the error message will be printed on the Debug screen every two seconds until the BASIC Stamp is reset. Simple DO...LOOP loops can be terminated with EXIT.

IF…THEN WITH A SINGLE *STATEMENT*

In addition to supporting everything discussed above, PBASIC 2.5 provides enhancements to the IF…THEN command that allow for more powerful, structured programming. In prior examples we've only used the first syntax form of this command: IF *Condition(s)* THEN *Address*. That form, while handy in some situations, can be quite limiting in others. For example, it is common to need to perform a single instruction based on a condition. Take a look at the following code:

```
' {$PBASIC 2.5}

x  VAR  Byte

FOR x = 1 TO 20                          ' count to 20
  DEBUG CR, DEC x                        ' display num
  IF (x // 2) = 0 THEN DEBUG " EVEN"     ' even num?
NEXT
```

This example prints the numbers 1 through 20 on the screen but every even number is also marked with the text " EVEN." The IF…THEN command checks to see if *x* is even or odd and, if it is even (i.e.: x // 2 = 0), then it executes the statement to the right of THEN: DEBUG " EVEN." If it was odd, it simply continued at the following line, NEXT.

Suppose you also wanted to mark the odd numbers. You could take advantage of the optional ELSE clause, as in:

```
' {$PBASIC 2.5}

x  VAR  Byte

FOR x = 1 TO 20                          ' count to 20
  DEBUG CR, DEC x
  IF (x // 2) = 0 THEN DEBUG " EVEN" ELSE DEBUG " ODD"
NEXT
```

This example prints the numbers 1 through 20 with " EVEN" or " ODD" to the right of each number. For each number (each time through the loop) IF…THEN asks the question, "Is the number even?" and if it is it executes DEBUG " EVEN" (the instruction after THEN) or, if it is not even it executes DEBUG " ODD" (the instruction after ELSE). It's important to note that this form of IF…THEN always executes code as a result of *Condition(s)*; it either does "this" (THEN) or "that" (ELSE).

INPUT

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

1 All 2

INPUT *Pin*

1
NOTE: Expressions are not allowed as arguments on the BS1. The range of the PIN argument on the BS1 is 0 – 7.

Function

Make the specified pin an input.

- ***Pin*** is a variable/constant/expression (0 – 15) that specifies which I/O pin to set to input mode.

Quick Facts

Table 5.41: INPUT Quick Facts.

	BS1	All BS2 Models
Input Pin Variables	PINS; PIN0 through PIN7	INS; IN0 through IN15
Related Commands	OUTPUT and REVERSE	

Explanation

There are several ways to make a pin an input. When a program begins, all of the BASIC Stamp's pins are inputs. Commands that rely on input pins, like PULSIN and SERIN, automatically change the specified pin to input. Writing 0s to particular bits of the variable DIRS makes the corresponding pins inputs. And then there's the INPUT command.

When a pin is an input, your program can check its state by reading the corresponding INS variable (PINS on the BS1). For example:

All 2

```
INPUT 4

Hold:
  IF IN4 = 0 THEN Hold                  ' stay here until P4 = 1
```

The code above will read the state of P4 as set by external circuitry. If nothing is connected to P4, it will alternate between states (1 or 0) apparently at random.

What happens if your program writes to the OUTS bit (PINS bit on the BS1) of a pin that is set up as an input? The value is stored in OUTS (PINS on the BS1), but has no effect on the outside world. If the pin is changed to output, the last value written to the corresponding OUTS bit (or PINS bit

```
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.

Demo Program (LCDIN.bsp)

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

```
' LCDIN.bsp
' This program demonstrates initialization, printing and reading
' from a 2 x 16 character LCD display.

' {$STAMP BS2p}
' {$PBASIC 2.5}

#IF ($STAMP < BS2P) #THEN
  #ERROR "Program requires BS2p, BS2pe or BS2px."
#ENDIF

Lcd             PIN     0

LcdCls          CON     $01     ' clear the LCD
LcdHome         CON     $02     ' move cursor home
LcdCrsrL        CON     $10     ' move cursor left
LcdCrsrR        CON     $14     ' move cursor right
LcdDispL        CON     $18     ' shift chars left
LcdDispR        CON     $1C     ' shift chars right
LcdDDRam        CON     $80     ' Display Data RAM
LcdCGRam        CON     $40     ' Character Generator RAM
LcdLine1        CON     $80     ' DDRAM address of line 1
LcdLine2        CON     $C0     ' DDRAM address of line 2

char            VAR     Byte(16)

Init_LCD:
  PAUSE 1000                    ' allow LCD to self-initialize first
  LCDCMD Lcd, %00110000         ' send wakeup sequence to LCD
  PAUSE 5                       ' pause required by LCD specs
  LCDCMD Lcd, %00110000
  PAUSE 0                       ' pause required by LCD specs
  LCDCMD Lcd, %00110000
  PAUSE 0                       ' pause required by LCD specs
  LCDCMD Lcd, %00100000         ' set data bus to 4-bit mode
  LCDCMD Lcd, %00101000         ' set to 2-line mode with 5x8 font
  LCDCMD Lcd, %00001100         ' display on without cursor
  LCDCMD Lcd, %00000110         ' auto-increment cursor

Main:
  DO
    LCDOUT Lcd, LcdCls, ["Hello!"]
    GOSUB Read_LCD_Screen
    PAUSE 3000
    LCDOUT Lcd, LcdCls, ["I'm a 2x16 LCD!"]
    GOSUB Read_LCD_Screen
    PAUSE 3000
  LOOP
```

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 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]
```

Table 5.77: Special Purpose Scratch Pad RAM Locations.

Location	BS2p and BS2pe
127	Bits 0-3, Active program slot #. Bits 4-7, program slot for READ and WRITE operations.
128	Polled input trigger status of Main I/O pins 0-7 (0 = not triggered, 1 = triggered).
129	Polled input trigger status of Main I/O pins 8-15 (0 = not triggered, 1 = triggered).
130	Polled input trigger status of Auxiliary I/O pins 0-7 (0 = not triggered, 1 = triggered).
131	Polled input trigger status of Auxiliary I/O pins 8-15 (0 = not triggered, 1 = triggered).
132	Bits 0-3: Polled-interrupt mode, set by POLLMODE
133	Bits 0-2: Polled-interrupt "run" slot, set by POLLRUN.
134	Bit 0: Active I/O group; 0 = Main I/O, 1 = Auxiliary I/O.
135	Bit 0: Polled-output status (set by POLLMODE); 0 = disabled, 1= enabled. Bit 1: Polled-input status; 0 = none defined, 1 = at least one defined. Bit 2: Polled-run status (set by POLLMODE); 0 = disabled, 1 = enabled. Bit 3: Polled-output latch status; 0 = real-time mode, 1 = latch mode. Bit 4: Polled-input state; 0 = no trigger, 1 = triggered. Bit 5: Polled-output latch state; 0 = nothing latched, 1 = signal latched. Bit 6: Poll-wait state; 0 = No Event, 1 = Event Occurred. (Cleared by POLLMODE only). Bit 7: Polling status; 0 = not active, 1 = active.

Demo Program (POLL.bsp)

NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the $STAMP directive accordingly.

```
' POLL.bsp
' This program demonstrates POLLIN, POLLOUT, and the use of the POLLMODE
' instruction.  Connect active-low inputs to pins 0, 1, 2, and 3.  Then
' connect an LED to pin 7. The program will print "." to the Debug
' window until one of the alarm buttons are pressed.  This will cause
' the termination of the main loop.  At this point the program will
' save the latched bits, clear them (and the polling process), then
' report the input(s) that triggered the alarm.

' {$STAMP BS2p}
' {$PBASIC 2.5}


FDoor           PIN     0
BDoor           PIN     1
Patio           PIN     2
Rst             PIN     3
AlarmLed        PIN     7

alarms          VAR     Byte            ' alarm bits
idx             VAR     Nib             ' loop control
```

```
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.

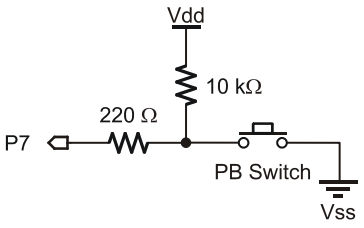

Figure 5.32: RANDOM Button Circuit.

2) **Serial input:** 123 (with no characters following it)
   Result: The BASIC Stamp halts at the SERIN command. It recognizes the characters "1", "2" and "3" as the number one hundred twenty three, but since no characters follow the "3", it waits continuously, since there's no way to tell whether 123 is the entire number or not.

3) **Serial input:** 123 (followed by a space character)
   Result: Similar to example 2, above, except once the space character is received, the BASIC Stamp knows the entire number is 123, and stores this value in *serData*. The SERIN command then ends, allowing the next line of code, if any, to run.

4) **Serial input:** 123A
   Result: Same as example 3, above. The "A" character, just like the space character, is the first non-decimal text after the number 123, indicating to the BASIC Stamp that it has received the entire number.

5) **Serial input:** ABCD123EFGH
   Result: Similar to examples 3 and 4 above. The characters "ABCD" are ignored (since they're not decimal text), the characters "123" are evaluated to be the number 123 and the following character, "E", indicates to the BASIC Stamp that it has received the entire number.

For examples of all formatters and how they process incoming data, see Appendix C.

WATCH OUT FOR ROLLOVER ERRORS.

Of course, as with all numbers in the BASIC Stamp, the final result is limited to 16 bits (up to the number 65535). If a number larger than this is received by the decimal formatter, the end result will look strange because the result rolled-over the maximum 16-bit value.

The BS1 is limited to the decimal formatter shown above, however all the BS2 models have many more conversion formatters available for the SERIN command. If not using a BS1, see the "Additional Conversion Formatters" section below for more information.

serial protocol is commonly called Synchronous Peripheral Interface (SPI) and is used by controller peripherals like ADCs, DACs, clocks, memory devices, etc.

At their heart, synchronous-serial devices are essentially shift-registers; trains of flip-flops that receive data bits in a bucket brigade fashion from a single data input pin. Another bit is input each time the appropriate edge (rising or falling, depending on the device) appears on the clock line.

SHIFTOUT OPERATION.

The SHIFTOUT instruction first causes the clock pin to output low and the data pin to switch to output mode. Then, SHIFTOUT sets the data pin to the next bit state to be output and generates a clock pulse. SHIFTOUT continues to generate clock pulses and places the next data bit on the data pin for as many data bits as are required for transmission.

Making SHIFTOUT work with a particular device is a matter of matching the mode and number of bits to that device's protocol. Most manufacturers use a timing diagram to illustrate the relationship of clock and data. One of the most important items to look for is which bit of the data should be transmitted first; most significant bit (MSB) or least significant bit (LSB). Table 5.117 shows the values and symbols available for *Mode* and Figure 5.43 shows SHIFTOUT's timing.

Symbol	Value	Meaning
LSBFIRST	0	Data is shifted out lsb-first
MSBFIRST	1	Data is shifted out msb-first

(Msb is most-significant bit; the highest or leftmost bit of a nibble, byte, or word. Lsb is the least-significant bit; the lowest or rightmost bit of a nibble, byte, or word.)

Table 5.117: SHIFTOUT Mode Values and Symbols.

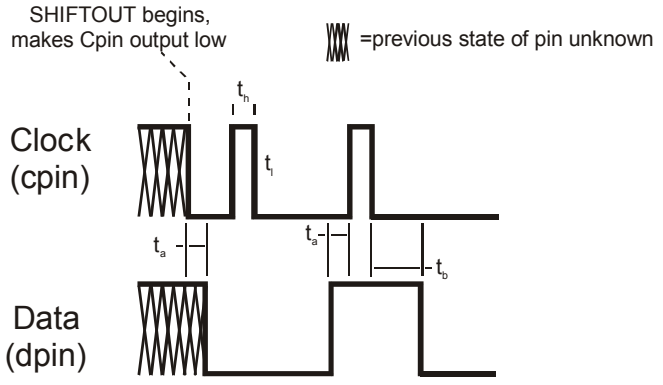

Figure 5.43: SHIFTOUT Timing Diagram. Refer to the SHIFTOUT Quick Facts table for timing information on t_h, t_l, t_a and t_b.

Demo Program (STORE0.bsp)

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

```
' STORE0.bsp
' This program demonstrates the STORE command and how it affects the READ
' and WRITE commands.  This program "STORE0.BSP" is intended to be down-
' loaded into program slot 0.  It is meant to work with STORE1.BSP and
' STORE2.BSP.  Each program is very similar (they display the current
' Program Slot and READ/WRITE Slot numbers and the values contained in the
' first five EEPROM locations.  Each program slot will have different data
' due to different DATA commands in each of the programs downloaded.

' {$STAMP BS2p, STORE1.BSP, STORE2.BSP}
' {$PBASIC 2.5}

#IF ($STAMP < BS2P) #THEN
  #ERROR "This program requires BS2p, BS2pe, or BS2px."
#ENDIF

idx             VAR     Word            ' index
value           VAR     Byte

LocalData       DATA    @0, 1, 2, 3, 4, 5


Main:
  GOSUB Show_Slot_Info                  ' show slot info/data
  PAUSE 2000
  STORE 1                               ' point READ/WRITE to Slot 1
  GOSUB Show_Slot_Info
  PAUSE 2000
  RUN 1                                 ' run program in Slot 1
  END

Show_Slot_Info:
  GET 127, value
  DEBUG CR, "Pgm Slot: ", DEC value.NIB0,
        CR, "R/W Slot: ", DEC value.NIB1,
        CR, CR

  FOR idx = 0 TO 4
    READ idx, value
    DEBUG "Location: ", DEC idx, TAB,
          "Value: ", DEC3 value, CR
  NEXT
  RETURN
```

STORE – BASIC Stamp Command Reference

All 2

Demo Program (TOGGLE.bs2)

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

```
' TOGGLE.bs2
' Connect LEDs to pins 0 through 3 as shown in the TOGGLE command descrip-
' tion in the manual and run this program. The TOGGLE command will treat
' you to a light show. You may also run the demo without LEDs. The Debug
' window will show you the states of pins 0 through 3.

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

thePin          VAR     Nib             ' pin 0 - 3


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

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