



Column #124, August 2005 by Jon Williams:

A BS2px ADC Trick and a BS1 Controller Treat

I love to travel. Yes, it has its inconveniences, but every time I feel the power of the jet I'm seated in lift off the runway a smile crosses my face. The downside of travel is, of course, living without the conveniences of home. For me that includes my stock of electronic parts for experimenting. I do carry a few things when I travel, but just enough to keep me entertained and keep the good folks at the TSA from getting nervous when they inspect my bags. So, as I sit here in my hotel room, I have my trusty PDB, a shiny new BS2px module, and the ubiquitous photocell. Let's see what we can cook up, shall we?

The BS2px is the latest edition to the BASIC Stamp microcontroller line. In addition to increased speed, the BS2px adds two new commands that give the programmer access to features available in the core SX microcontroller: CONFIGPIN and COMPARE.

CONFIGPIN is somewhat similar to manipulating the DIRS register in that it configures I/O pins, but it has four independent modes that allow for advanced pin behavior.

CONFIGPIN mode 0 (SCHMITT) sets the I/O pin to Schmitt trigger mode, causing hysteresis to be added to the inputs. This can be very useful if the input of a pin is a bit noisy and

hovering around the normal TTL (1.4 volt) or CMOS (2.5 volt – see below) threshold level. When set to Schmitt mode, a pin will transition from 0-to-1 when it crosses above about 85% of Vdd, or about 4.25 volts. A pin will transition from 1-to-0 when it crosses below about 15% of Vdd, or about 0.75 volts.

Here's how we can set P15 as Schmitt trigger input:

```
CONFIGPIN SCHMITT, %1000000000000000
```

How might we use this mode? Well, one thing we could do with it is clean up slow-rising or slow-falling waveforms. Combined with COUNT or PULSIN, one could measure the frequency of a sine wave input (Note: make sure the input to the pin does not go below Vss).

CONFIGPIN mode 1 (THRESHOLD) sets pin input threshold. On reset, all pins are configured for TTL threshold level: 1.4 volts. We can raise this up to 2.5 volts (50% of Vdd) by configuring the pins as CMOS inputs. Note that the CMOS threshold will change the behavior of some commands. RCTIME, for example, will return a smaller value when using the “standard” circuit and measuring while a pin is high because the span between five volts and 2.5 volts (CMOS) is smaller than the span between five volts and 1.4 volts (TTL). This can actually be helpful when we have a very large “R” constituent in RCTIME.

Run this little program with a standard RCTIME circuit (Figure 124.1) to see the difference between the two modes. Figure 2 shows the program output.

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

Main:
  DO
    CONFIGPIN THRESHOLD, %0000000000000000
    GOSUB Read_Pot
    DEBUG CRSRXY, 8, 2, DEC potVal, CLREOL

    CONFIGPIN THRESHOLD, %1000000000000000
    GOSUB Read_Pot
    DEBUG CRSRXY, 8, 3, DEC potVal, CLREOL

  PAUSE 100
  LOOP
Read_Pot:
  HIGH PotPin
  PAUSE 1
  RCTIME PotPin, 1, potVal
  RETURN
```

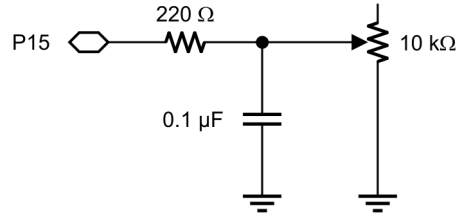


Figure 124.1: Standard RCTIME Circuit

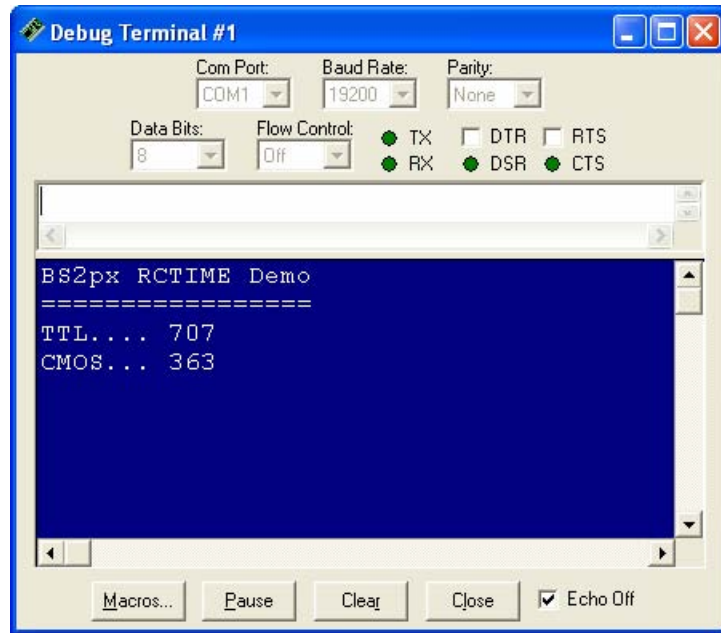


Figure 124.2: Program Output

CONFIGPIN mode 2 (PULLUP) allows us to enable ~20K pull-up resistors on selected IO pins. This can simplify external circuit design if we can live with active-low inputs (remember that we can easily make an active-low input look active-high using the invert operator).

Column #124: A BS2px ADC Trick and a BS1 Controller Treat

Finally, CONFIGPIN mode 3 (DIRECTION) sets the pin direction – very much the same as modifying the DIRS register. The following lines of code perform the exactly the same function:

```
DIRS = %00000000000001111
CONFIGPIN DIRECTION, %00000000000001111
```

Both lines set pins P0 - P3 to outputs; the difference being that the second line only works on the BS2px module.

The second new command that the BS2px offers is called COMPARE. This allows us to enable and use the onboard comparator. All SX micros have a comparator with inputs on RB.1 and RB.2; these pins map to P1 and P2 on the BS2px. Depending on configuration, the output of the comparator can be routed to RB.0 which maps to BS2px P0.

One of the things that BASIC Stamp users frequently wish for is an ADC – even a simple eight-bit ADC will do in many applications. Well, we can use the BS2px comparator, an addition output pin, and a couple RC components to make one. Really.

In fact, my old pal Scott Edwards (for those of you that are new to Nuts & Volts, Scott is the originator of the “Stamp Applications” column) did this using a hardware comparator and the BS1 (see column #25 – available for download from Parallax and from Nuts & Volts). The process is actually pretty simple: we will use the PWM instruction to generate a voltage (through the RC network) that feeds one side of the comparator. The other side of the comparator will be the input for the voltage we want to measure (0 to 5 volts). When the voltage we generate crosses the level that we’re measuring we can detect a change in the comparator output and know that we’ve reached the input voltage. Figure 124.3 shows the connections.

The code is equally simple: the subroutine that measures the input basically “sneaks up” on the input voltage by running a loop until the comparator output says we’ve crossed the input level.

```
Get_ADC:
  FOR adcVal = 0 TO 255
    PWM DacOut, adcVal, 1
    COMPARE 2, result
    IF (result = 1) THEN EXIT
  NEXT
RETURN
```

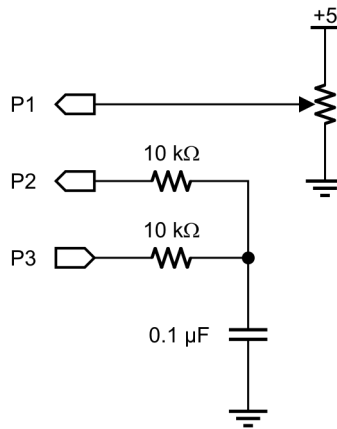


Figure 124.3: Program Output

In Scott's original article he included another version that used a binary search method to speed up the conversion, but with the speed of the BS2px it's not necessary to get that fancy. I did give it a try, but found that it was a bit noisy – perhaps due to components on my PDB. That said, I never saw any ADC input fluctuations using the subroutine above so that's what I'm going to stick with.

To fancy things up a bit, though, the main code converts the ADC reading to millivolts. With a five volt input, each bit is equal to 19.6 millivolts ($5.00 / 255 = 0.0196$). We can use the `*/` (star-slash) operator to do the fractional multiplication for us and get the result as shown in Figure 124.4.

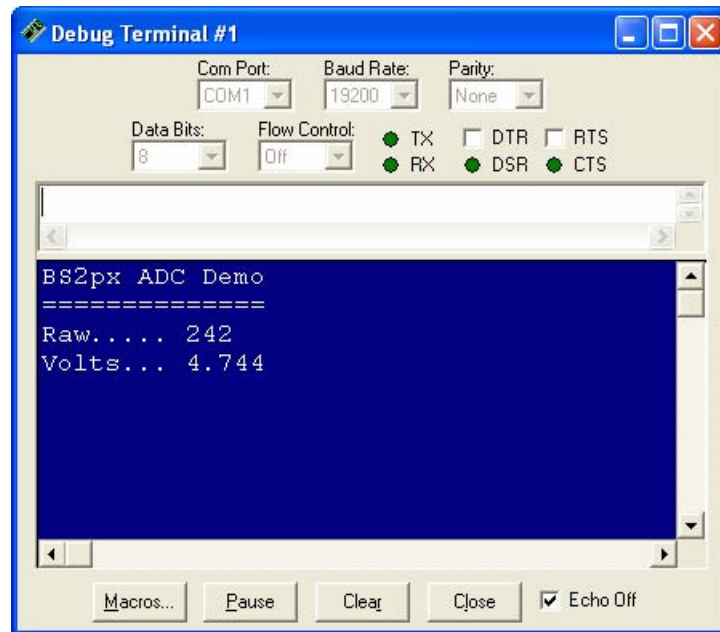


Figure 124.4: Program Output in Debug Terminal

```
Main:
DO
  GOSUB Get_ADC
  mVolts = adcVal */ $139B
  DEBUG CRSRXY, 9, 2,
    DEC adcVal, " ",
    CRSRXY, 9, 3,
    DEC1 (mVolts / 1000), ".",
    DEC3 mVolts
  PAUSE 100
LOOP
```

So, we end up using as many pins as if we installed the ADC0831, but with fewer parts and fairly simple code. It's a nice trick to have in a pinch.

In this program we're not having P0 track with the comparator output, but it can be done (using mode 1). This is especially useful if we're using pin polling – by configuring P0 as a

polled pin, the comparator can cause an action automatically. We could, for example, use POLLRUN to switch to another program slot based on a change in the comparator status.

One of the things that I frequently remind my friends to do is have a look at the BASIC Stamp Editor's Help file. Please do that – it's the most up-to-date source of information for BASIC Stamp modules. The BS2px is the fastest of the BS2 family, so there are a few commands with syntax changes which may force us to update our programs (notably SERIN and SEROUT). Remember, conditional compilation is always available and will let us move easily between the BS2px and other BASIC Stamp models (except where CONFIGPIN and COMPARE are used). In fact, both BS2px demos this month have the following bit of code at the top of the program:

```
Check_Stamp:
  #IF ($STAMP <> BS2PX) #THEN
    #ERROR "This program requires the BS2px"
  #ENDIF
```

Since CONFIGPIN and COMPARE are only available on the BS2px, this will cause a dialog that alerts us to change the BASIC Stamp module if we have the wrong type installed.

Custom Prop Control Made Easy

One of the reasons I'm currently traveling is to work with a group of folks who enjoy building Halloween and other holiday props/displays as a hobby. Interesting group if I do say so myself. All kidding aside, I've found the people most steeped in the scary stuff of Halloween to be as "normal" as your next-door neighbor – in fact, they just might be your next-door neighbor, albeit with an interestingly dark hobby.

Some of these folks are new to programming and working with customized controllers, so what my colleague, John Barrowman, and I have been doing is showing them how they can take a general-purpose controller like the BASIC Stamp and make it behave like a purpose-built prop controller (that often costs more money and is fixed at one behavior).

To give you an example, a simple prop timer will wait for an input, create a delay before activating the prop-control output, hold the output on for a period of time, then turn the output off and hold it off for a while so the prop cannot be immediately retriggered. One of the most popular off-the-shelf products that fits this description is called the *Universal Dual Timer II* (UDT2) by a company called Terror By Design (trust me, Denny and his crew are really nice people, despite the scary company name). The UDT2 is very popular amongst Halloween prop builders in that all of the timing is set with simple knobs.

One of my recent prop project “show-and-tells” was a light activated prop controller that works like a digital version of the UDT2. It uses a photocell to measure light as the trigger, and even has a random timing feature that can give a prop more dynamic behavior. Let’s build it, shall we?

Since the requirements are so simple, we’ll use a BS1 for the prop controller. Start by creating the light sensor circuit shown in Figure 124.5.

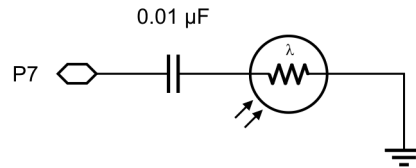


Figure 124.5: Light Sensor Circuit

Note that the RC circuit for the BS1’s POT instruction is configured differently than RC circuits used with the BS2’s RCTIME. The reason for this is that POT is actually an active command: it makes the pin an output, charges the capacitor, then actively discharges and checks the capacitor voltage level to do the reading. Stamp Applications column #15 (also written by Scott Edwards) gives a detailed explanation of POT versus RCTIME. Also note the small value of the capacitor in the circuit; this is necessary due to the very large dark resistance of the CdS photocell.

The thing about POT that makes it a bit trickier to use than RCTIME is that it requires a calibration constant in the syntax. The purpose of this constant is to scale the resulting RC measurement to a maximum of 255 so that it will fit into a byte (remember, the BS1 doesn’t have much memory). While we could derive the scale value empirically, there’s a tool built into the Editor that will do it for us.

From the Run menu select Prop Scaling. This will open a small dialog that lets us set the pin connected to the RC circuit. After that selection is made, click Start and the Editor will download a small program to the BS1 (yes, this overwrites the program that was previously there). What we want to do now is adjust the RC circuit for the smallest Scale value.

The way we adjust the CdS photocell circuit is by shielding it from light. Using one of the larger sensors out of a CdS multi-pack from RadioShack® I found that the Scale value was 159.

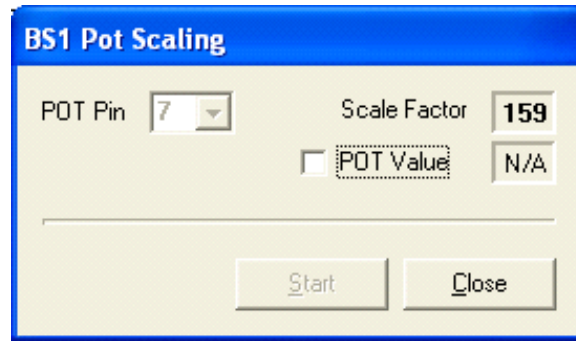


Figure 124.6: BS1 Pot Scaling Dialog Box

Figure 6 shows the Pot Scaling dialog box. To test the Scale value, click on the POT Value checkbox and watch the value change as the light falling on the sensor changes. What we'd like to have is a nice range from 0 (very bright) to 255 (dark). Why do we get a small value when the light is bright? Well, a photocell is a light-dependent resistor, and the resistance is inversely proportional to the amount of light falling on it.

What we're going to do for the prop controller is monitor the input and when the light falling on the sensor drops (reading goes up), we'll trigger the prop. In this mode we can detect the presence of a "victim" that blocks the light reaching the photocell.

The first thing to do is measure the ambient light and set a threshold. Here's how we do it:

```
POT LSense, 159, thresh
thresh = thresh * 12 / 10
```

The POT instruction reads the current light level, and then the next line sets the threshold to 120% of that reading. By setting threshold higher than the original reading we ensure that a real change is made before the prop trips (we're adding hysteresis to the light detection logic). The nice thing about auto-calibrating in software is that we can recalibrate the prop any time we want simply by resetting the BASIC Stamp module.

Column #124: A BS2px ADC Trick and a BS1 Controller Treat

With the threshold set we drop to the main loop of code that waits for the light hitting the sensor to drop.

```
Main:
  RANDOM rndVal
  POT LSense, 159, light
  IF light < thresh THEN Main
```

Notice that the RANDOM function is called during this loop too. What this does is stir the BASIC Stamp's pseudo-random number generator until we get a trigger level input. This is a great way to add true randomness to a program (since we cannot predict when the prop will be triggered).

With the prop triggered, the first timing function is the pre-event delay, and this is the time we want to make somewhat random. I say somewhat random because we understandably want the time to fall within a given range.

```
Sequence:
  mnTimer = DlyMin
  mxTimer = DlyMax
  GOSUB Random_Timer
```

In order to keep the program easy to maintain, the constants section will hold all the prop timing values. For the pre-event delay we will use a random time between DlyMin and DlyMax. Let's drop down to the timing code:

```
Random_Timer:
  span = mxTimer - mnTimer + 1
  secs = rndVal // span + mnTimer

Timer:
  IF secs = 0 THEN Timer_Done
  PAUSE 1000
  secs = secs - 1
  GOTO Timer

Timer_Done:
  RETURN
```

This subroutine actually has two entry points: the top, at Random_Timer randomizes the value of secs based on the current value of the random number generator (in rndVal) and the values passed in mnTimer and mxTimer. This code also takes advantage of the modulus (//)

operator. By dividing the random value by the span plus one (of our timing extremes) we get a number between zero and the span. When we add the minimum timing value the result in secs will be between those passed in mnTimer and mxTimer.

Let's work through an example: Perhaps we want a randomized delay between five and 10 seconds. The span is calculated as 10 minus five plus 1 (six). Since the modulus operator returns the remainder of a division, using six as the divisor will give us a value between zero and five. By adding our minimum value of five we finally end up with a value between our targets of five and 10.

The value in secs is used as a counter for the subroutine entry at Timer. This is pretty simple; while secs is greater than zero we will PAUSE for one second (1000 milliseconds), decrement secs, and then check it again. When the value of secs does reach zero the subroutine will terminate and we'll return to the main program.

With the randomized timing done the rest of the prop control sequence is fairly simple.

```

Prop = IsOn
secs = OnTime
GOSUB Timer
Prop = IsOff

secs = OffTime
GOSUB Timer

GOTO Main
    
```

As you can see, there is nothing mysterious at all about this. We simply activate the prop control output, start the timer, deactivate the prop output, then run the timer for the required down-time before the next possible trigger event.

Let me point out again that the program uses defined symbols so that any changes we want to make all happen in the same place in the code. Here are the constant declarations for the prop timer:

```

SYMBOL DlyMin           = 5
SYMBOL DlyMax           = 15
SYMBOL OnTime           = 5
SYMBOL OffTime          = 30
    
```

This version of the program uses a timing resolution of one second. In some cases this may be a bit coarse. If more resolution is required, it's easy enough to change secs to a Word, and update the PAUSE statement in the Timer subroutine to 100 milliseconds. Since there's not quite enough variable space left to pass fraction seconds in mnTimer and mxTimer, we'll also need to add a line to the end of the randomizing section of the timer routine.

```
secs = secs * 10
```

This will correct the value for the increased resolution.

Using Photocells with the Prop-1 Controller

If you connect the circuit shown in Figure 124.5 to a Prop-1 controller you'll find that it doesn't behave as you expect. What gives? Well, all of the I/O pins on the Prop-1 are connected to the input side of a ULN2803 driver – the driver is preventing the **POT** instruction from working properly. We have a few choices to correct this.

The first is the easiest – simply remove the ULN2803 from its socket. This choice is only viable, though, if we're using the TTL outputs from the Prop-1 and have no need for the high-current side (See Figure 124.7, top diagram).

The second choice is to put the POT circuit on P7, and then replace the ULN2803 with a ULN2003. When doing this the ULN2003 must be inserted such that it is bottom-aligned with the ULN2803 socket. This will open the connection to the P7 circuit (See Figure 124.7, middle diagram). Note that when using a POT circuit on P7 or P6 of the Prop-1 controller, you must remove the input configuration (SETUP) jumper on the pin being used.

Finally, and the least desirable, is to do a bit of “surgery” on the ULN2803. Use this choice only if the first two are not available. Remove the ULN2803 and cut off the input and output legs that correspond to the pin that you'll use. Note that ULN2803 pins 1 and 18 correspond to P7, ULN2803 pins 2 and 17 correspond to P6, and so on. Do not remove ULN2803 pins 9 or 10 – this will prevent the high-current outputs from working properly. The bottom diagram in Figure 124.7 shows a ULN2803 modified to allow a POT circuit to work on P0. Again, do this only as your last resort, and make sure you have a spare ULN2803 or two before you start “operating.”

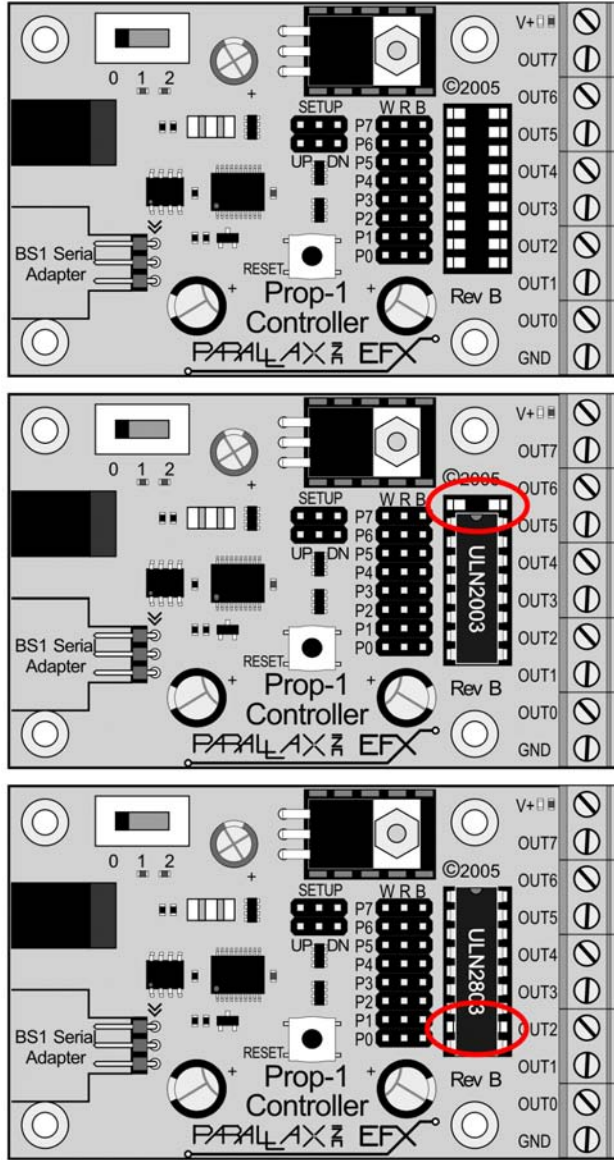


Figure 124.7: Prop-1 Controller and ULN2803 Options

Column #124: A BS2px ADC Trick and a BS1 Controller Treat

Okay, I'd say that's about enough from the road. Halloween is just a couple months away – you'd better get busy with those props!

Until next time – when I'm home and comfy – Happy Stamping.

Additional Resources

Terror by Design
www.terrorbydesign.com

```

' =====
'
' File..... Schmitt_RCTIME.BPX
' Purpose.... Demonstrates Schmitt level effect on RCTIME
' Author..... Jon Williams -- Parallax, Inc.
' E-mail..... jwilliams@parallax.com
' Started....
' Updated.... 18 JUN 2005
'
'   {$STAMP BS2px}
'   {$PBASIC 2.5}
' =====
'
' -----[ Program Description ]-----
'
' -----[ Revision History ]-----
'
' -----[ I/O Definitions ]-----
PotPin          PIN      15
'
' -----[ Constants ]-----
'
' -----[ Variables ]-----
potVal          VAR      Word
'
' -----[ EEPROM Data ]-----
'
' -----[ Initialization ]-----
Check_Stamp:
  #IF ($STAMP <> BS2PX) #THEN
    #ERROR "This program requires the BS2px"
  #ENDIF
Setup:
  DEBUG CLS,
    "BS2px RCTIME Demo", CR,
    "=====", CR,
    "TTL...           ", CR,
    "CMOS...          "

```

Column #124: A BS2px ADC Trick and a BS1 Controller Treat

```
' -----[ Program Code ]-----  
Main:  
DO  
  CONFIGPIN THRESHOLD, %0000000000000000  
  GOSUB Read_Pot  
  DEBUG CRSRXY, 8, 2, DEC potVal, CLREOL  
  
  CONFIGPIN THRESHOLD, %1000000000000000  
  GOSUB Read_Pot  
  DEBUG CRSRXY, 8, 3, DEC potVal, CLREOL  
  
  PAUSE 100  
LOOP  
  
END  
  
' -----[ Subroutines ]-----  
Read_Pot:  
  HIGH PotPin  
  PAUSE 1  
  RCTIME PotPin, 1, potVal  
  RETURN
```

```

' =====
'
' File..... SW20-EX35-BS2px-ADC.BPX
' Purpose.... Serial communications with PC
' Author..... (C) 2000 - 2005, Parallax, Inc.
' E-mail..... support@parallax.com
' Started....
' Updated.... 15 JUL 2005
'
' {$STAMP BS2px}
' {$PBASIC 2.5}
' =====

' ----[ Program Description ]-----
'
' Creates a simple 8-bit ADC with the BS2px using the internal comparator.

' ----[ Revision History ]-----

' ----[ I/O Definitions ]-----

Vin          PIN      1          ' unknown voltage input
DacIn        PIN      2          ' input from R/C DAC
DacOut       PIN      3          ' DAC via PWM + R/C

' ----[ Constants ]-----

' ----[ Variables ]-----

adcVal       VAR      Byte       ' adc value (0 - 255)
bias         VAR      Byte       ' bias for ADC conversion
compOut      VAR      Bit        ' comparator result bit
mVolts       VAR      Word       ' input in millivolts

' ----[ EEPROM Data ]-----

' ----[ Initialization ]-----

Check_Stamp:
  #IF ($STAMP <> BS2PX) #THEN
    #ERROR "This program requires the BS2px"
  #ENDIF

```

Column #124: A BS2px ADC Trick and a BS1 Controller Treat

```
Setup:
  DEBUG CLS,
    "BS2px ADC Demo", CR,
    "=====", CR,
    "Raw.....", CR,
    "Volts..."

' -----[ Program Code ]-----

Main:
DO
  GOSUB Get_ADC          ' read comparator ADC
  mVolts = adcVal * / $139B ' convert to millivolts

  DEBUG CRSRXY, 9, 2,          ' show results
    DEC adcVal, " ",
    CRSRXY, 9, 3,
    DEC1 (mVolts / 1000), ".",
    DEC3 mVolts

  PAUSE 250
LOOP

' -----[ Subroutines ]-----

Get_ADC:
  adcVal = 0              ' clear ADC
  bias = 128              ' start in middle
DO
  adcVal = adcVal + bias  ' add bias to adc result
  PWM DacOut, adcVal, 15 ' output new test value
  COMPARE 2, compOut      ' check comparator
  IF (compOut = 1) THEN   ' if unknown lower than test
    adcVal = adcVal - bias ' -- reduce adcVal
  ENDIF
  bias = bias / 2         ' check next half
LOOP UNTIL (bias = 0)
RETURN

Get_ADC_Simple:
  adcVal = 0              ' clear ADC
DO
  PWM DacOut, adcVal, 15 ' output new value
  COMPARE 2, compOut      ' check comparator
  IF (compOut = 1) THEN EXIT ' voltage found
  adcVal = adcVal + 1     ' increment result
LOOP UNTIL (adcVal = 255)
RETURN
```

```

' =====
'
' File..... Prop_Timer.BS1
' Purpose.... Simulate simple single output prop timer with BASIC Stamp
' Author..... Jon Williams -- Parallax, Inc.
' E-mail..... jwilliams@parallax.com
' Started....
' Updated.... 19 JUN 2005
'
' {$STAMP BS1}
' {$PBASIC 1.0}
' =====

' ----[ Program Description ]-----
'
' This program simulates a simple single-output program control/timer, and
' adds the ability to randomize the delay time between the trigger input
' and prop activation.

' ----[ Revision History ]-----

' ----[ I/O Definitions ]-----

SYMBOL LSense          = 7           ' Light sensor input
SYMBOL Prop            = PIN0        ' Prop control output

' ----[ Constants ]-----

SYMBOL IsOn            = 1           ' active high
SYMBOL IsOff           = 0

' *****
' Adjust these values to control prop
' *****

SYMBOL DlyMin          = 5           ' minimum delay
SYMBOL DlyMax          = 15          ' maximum delay
SYMBOL OnTime          = 5           ' prop on time
SYMBOL OffTime         = 30          ' prop off time

' ----[ Variables ]-----

SYMBOL thresh          = B0          ' light threshold
SYMBOL light           = B1          ' light level

```

Column #124: A BS2px ADC Trick and a BS1 Controller Treat

```
SYMBOL mnTimer      = B2      ' minimum timer value
SYMBOL mxTimer      = B3      ' maximum timer value
SYMBOL span         = B4      ' random timer range
SYMBOL secs         = B5      ' timer value

SYMBOL rndVal       = W3      ' random value

' -----[ EEPROM Data ]-----

' -----[ Initialization ]-----

Setup:
  POT LSense, 159, thresh      ' get initial light level
  thresh = thresh * 12 / 10    ' adjust to 120%

  DIRS = %00000001            ' make prop output and off
  rndVal = 1031                ' initialize seed

' -----[ Program Code ]-----

Main:
  RANDOM rndVal                ' stir random generator
  POT LSense, 159, light       ' get current light level
  IF light < thresh THEN Main  ' wait for light change

Sequence:
  mnTimer = DlyMin
  mxTimer = DlyMax
  GOSUB Random_Timer          ' do random delay

  Prop = IsOn                  ' activate prop
  secs = OnTime                ' - for "OnTime"
  GOSUB Timer
  Prop = IsOff                 ' deactivate prop

  secs = OffTime               ' load minimum off time
  GOSUB Timer

  GOTO Main                    ' back to top

' -----[ Subroutines ]-----

' Put minimum timer value in "mnTimer" and maximum timer value in "mxTimer"
' -- "mxTimer" must be larger than "mnTimer"
' -- secs will be calculated from those two values
' -- secs value drops through to Timer subroutine
```

```
Random_Timer:
  span = mxTimer - mnTimer + 1           ' get span
  secs = rndVal // span + mnTimer        ' get random secs in range

' Put number or seconds in "secs" before calling
' -- secs is modified by this subroutine
' -- maximum delay is 255 second (4 mins, 15 seconds)

Timer:
  IF secs = 0 THEN Timer_Done
  PAUSE 1000
  secs = secs - 1
  GOTO Timer

Timer_Done:
  RETURN
```

```
' =====
'
' File..... Prop_Timer_HR.BS1
' Purpose.... Simulate simple single output prop timer with BASIC Stamp
' Author..... Jon Williams -- Parallax, Inc.
' E-mail..... jwilliams@parallax.com
' Started....
' Updated.... 19 JUN 2005
'
'   {$STAMP BS1}
'   {$PBASIC 1.0}
' =====

' -----[ Program Description ]-----
'
' This program simulates a simple single-output program control/timer, and
' adds the ability to randomize the delay time between the trigger input
' and prop activation.
'
' This version increases timer resolution to 0.1 secs.

' -----[ Revision History ]-----

' -----[ I/O Definitions ]-----

SYMBOL  LSense      = 7           ' Light sensor input
SYMBOL  Prop        = PIN0       ' Prop control output

' -----[ Constants ]-----

SYMBOL  IsOn        = 1           ' active high
SYMBOL  IsOff       = 0

' *****
' Adjust these values to control prop
' *****

SYMBOL  DlyMin      = 5           ' minimum delay (secs)
SYMBOL  DlyMax      = 15          ' maximum delay
SYMBOL  OnTime      = 5           ' prop on time
SYMBOL  OffTime     = 30          ' prop off time

' -----[ Variables ]-----
```

```

SYMBOL thresh      = B0      ' light threshold
SYMBOL light       = B1      ' light level

SYMBOL mnTimer     = B2      ' minimum timer value
SYMBOL mxTimer     = B3      ' maximum timer value
SYMBOL span        = B4      ' random timer range
SYMBOL tenths      = W3      ' timer value

SYMBOL rndVal      = W4      ' random value

' -----[ EEPROM Data ]-----

' -----[ Initialization ]-----

Setup:
  POT LSense, 159, thresh      ' get initial light level
  thresh = thresh * 12 / 10    ' adjust to 120%

  DIRS = %00000001           ' make prop output and off
  rndVal = 1031               ' initialize seed

' -----[ Program Code ]-----

Main:
  RANDOM rndVal                ' stir random generator
  POT LSense, 159, light       ' get current light level
  IF light < thresh THEN Main  ' wait for light change

Sequence:
  mnTimer = DlyMin
  mxTimer = DlyMax
  GOSUB Random_Timer           ' do random delay

  Prop = IsOn                  ' activate prop
  tenths = OnTime              ' - for "OnTime"
  GOSUB Timer
  Prop = IsOff                  ' deactivate prop

  tenths = OffTime             ' load minimum off time
  GOSUB Timer

  GOTO Main                    ' back to top

' -----[ Subroutines ]-----

' Put minimum timer value in "mnTimer" and maximum timer value in "mxTimer"
' -- "mxTimer" must be larger than "mnTimer"

```

Column #124: A BS2px ADC Trick and a BS1 Controller Treat

```
' -- tenths will be calculated from those two values
' -- tenths value drops through to Timer subroutine

Random_Timer:
  span = mxTimer - mnTimer + 1           ' get span
  tenths = rndVal // span + mnTimer      ' get random secs in range
  tenths = tenths * 10                   ' change to 0.1 increments

' Put number or 0.1 sec intervals in "tenths" before calling
' -- tenths is modified by this subroutine
' -- maximum delay is 6553.5 seconds

Timer:
  IF tenths = 0 THEN Timer_Done
  PAUSE 100
  tenths = tenths - 1
  GOTO Timer

Timer_Done:
  RETURN
```