



Column #130, February 2006 by Jon Williams:

Lights, BASIC Stamp, Action!

Having been raised in the desert of southern California I'm pretty much a warm weather person. The fact is I don't like the winter months; not at all, nothing about them. Okay, except for one thing: holiday lights. The coolest thing about the Christmas season is the lights. I know what you're thinking: "Hey pal, that was two months ago." True, but we can do lighting any time of year – and not just for holiday displays. In fact, my own interest in lighting control started when I was 18 and working with a friend's rock-n-roll band. With the crush of the holiday season behind us, let's have a look at lighting control and strategies for getting the most from our code. Who knows, we just might come up with a way to light up our Valentine's life...



Author's Note: This article deals with products that can switch 120 VAC which, if mishandled, can be dangerous and even lethal. Your safety is in your own hands; if you have any doubts about working with 120 VAC circuits it is strongly suggested that you seek assistance from a qualified electrician.

Those that haven't tried it may think controlling a 120 VAC lamp is trivial – we do it all the time, right, we just flip a switch? But doing that with a microcontroller is a little more involved. The first thing that comes to mind is a relay, which works, but presents its own challenges. Challenge number one is that we usually need some sort of buffer (e.g., a ULN2803) between the microcontroller and the relay, and two, contact arcing can become a serious issue, especially when it leads to contact fusing. And, mechanical relays are just plain noisy. Who needs all that clickity-clack nonsense? I don't... do you?

Column #130: Lights, BASIC Stamp, Action!

Okay, then, what to do? Use a relay, of course; a solid state relay. A solid state relay (SSR) isn't actually a relay, it's a special circuit that [usually] takes a low-voltage, low-current input and switches a high-voltage AC or DC output – with optical isolation between the input and output sides. The great thing about SSRs is that we can easily find units that will connect directly to the BASIC Stamp and switch 120 VAC outputs. A popular SSR is the Crydom D2W series, like the one shown in Figure 130.1 (in my hand for scale). It draws just a few milliamps at five volts on the input side.



Figure 130.1: A Crydom DW2 Series SSR

Just before the last holiday season the Parallax EFX group released the RC-4 relay board that is designed to hold up to four Crydom D2W (and compatible) SSRs. The RC-4 takes a TTL serial command input and will activate the relays accordingly. The RC-4 is addressable so up to four boards (a total of 16 control outputs) are available on a single BASIC Stamp I/O pin. And, it has a baud setting jumper that allows it to work with the BS1 at 2400 baud, or with the BS2-family at 38.4 kBaud.

The RC-4 became quite popular with holiday decorators this past season, though some stumbled over using more than one in a single application. It's really pretty easy, and PBASIC2 gives us the ability to treat our control outputs (up to 16) as a single [Word] variable; we'll see how in a just a moment. And, by taking advantage of conditional compilation we can structure our lighting control program such that it can be configured to run on the Stamp CI board and use Opto-22 SSRs for the outputs. These products are widely used in industrial applications, and provide additional features like fuse-protected, high-voltage outputs.

Getting Conditional

In the past we've typically used conditional compilation to set program constants based on the BASIC Stamp module in use. We'll do that again, but we're also going to add a custom program switch so that we can decide on serial output for a network of RC-4 boards, or parallel output when using the program with a Stamp CI board and Opto-22 relays.

Let's start at the very top. Using #DEFINE we can create a conditional symbol that will affect other parts of the program.

```
#DEFINE __SerialMode = 1

#IF __SerialMode #THEN
  Sio          PIN      15
#ELSE
  Lights       VAR      OUTS
#ENDIF

Speed         PIN      14
PgmSelect     PIN      12
```

In order to keep conditional symbols straight in my own head I've decided to preface them with two underscore characters. As you can see, the first place we use the __SerialMode symbol is to define our output structure. An important thing to remember is that the only parts of the program that get compiled and downloaded are those that satisfy the conditional statement. In the code above, for example, the program doesn't ever know about the symbol called Lights because the setting of __SerialMode excludes it from compilation. Of course, if we want to switch to parallel mode (for use with the Stamp CI board) we would change the value of __SerialMode to zero.

There are two I/O pins common to both modes: one selects the output sequence (we're keeping this very simple with just two sequences), and the second is used to read a potentiometer with RCTIME so that we can control step speed when the program is running.

Moving into the body of the program we will use __SerialMode again, this time to set pins P0-P11 to outputs when we're configured for parallel mode:

```
Reset:
  #IF (__SerialMode = 0) #THEN
    DIRS = $0FFF
  #ENDIF
  lightVal = $0000
  GOSUB Update_Outputs
```

Column #130: Lights, BASIC Stamp, Action!

In either case, the state of the light outputs (in lightVal) is cleared and the Update_Outputs subroutine is called to initialize this state:

```
Update_Outputs:
  #IF __SerialMode #THEN
    SEROUT Sio, Baud, ["!RC4", %00, "S", lightVal.NIB0,
                      "!RC4", %01, "S", lightVal.NIB1,
                      "!RC4", %10, "S", lightVal.NIB2]
  #ELSE
    Lights = lightVal & $0FFF
  #ENDIF
  RETURN
```

And here's where we really get to the point of using __SerialMode as it directs the value of lightVal to the appropriate destination. This is a really good demonstration of the power and flexibility of PBASIC2. Since each RC-4 has four outputs (conveniently, the size of a nibble) we are able to use the NIB variable modifier to select the appropriate bits (used by the RC-4's Set command) for each board.

You may be wondering if this will work as we've squeezed three output commands into a single SEROUT statement. Yes, it does work. The reason it does is that each RC-4 is waiting for the "!RC4" header and its own board address. If the board address is wrong that RC-4 will go right back to waiting for the header (for those who are also interested in programming the SX with SX/B, the RC-4 control code is written entirely in SX/B with the same techniques we used last month with the PSX helper). By putting all three commands into a single SEROUT statement the program runs a little quicker (because we don't have to reload and configure additional SEROUT statements) – this will be important when we're running quick steps.

Just a couple tips on the RC-4: It actually gets its power from the serial line, so if you're going to connect it to a BOE servo header, you must make sure the servo power jumper is set to Vdd (+5 vdc). On the RC-4 you'll see two headers marked SER. This allows the RC-4 to be daisy-chained as shown in Figure 130.2. Note that the Baud jumper is inserted for 38.4k when using the BS2-family. If you're going to connect the RC-4 to a Prop-1, remove the Baud jumper for 2400 baud operation.

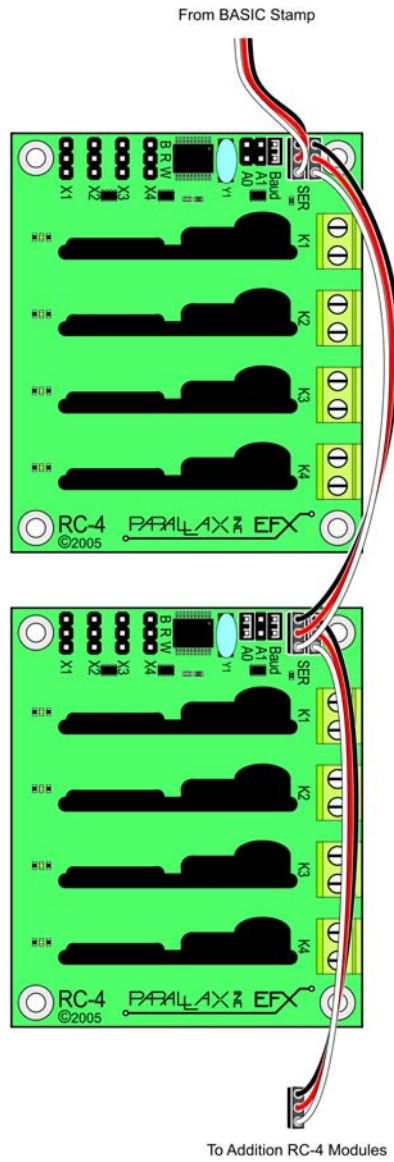


Figure 130.2: Daisy-Chained RC-4 Modules

Column #130: Lights, BASIC Stamp, Action!

As you can see in Figure 130.2, each RC-4 has two jumpers, A0 and A1, that allow the board to have a unique address (%00 to %11). When a jumper is inserted that address bit is set to one; removed clears the bit to zero. In Figure 130.2 the top RC-4 is set to address %00 (both jumpers removed) and the lower board set to %01 (A0 in, A1 out). The other nice thing about the RC-4 is that it conforms to the Parallax AppMod protocol, so you can put it on the same serial line as other serial devices, such as the Parallax Servo Controller (PSC).

Let's get back to running our light program. The sequences are stored in DATA statements like this:

```
Zig          DATA   Word  %000000000001
             DATA   Word  %000000000010
             DATA   Word  %000000000100
             DATA   Word  %000000001000
             DATA   Word  %000000010000
             DATA   Word  %000000100000
             DATA   Word  %000001000000
             DATA   Word  %000010000000
             DATA   Word  %000100000000
             DATA   Word  %001000000000
             DATA   Word  %010000000000
             DATA   Word  %100000000000

Wash         DATA   Word  %000000000000
             DATA   Word  %000001100000
             DATA   Word  %000011110000
             DATA   Word  %000111111000
             DATA   Word  %001111111100
             DATA   Word  %011111111110
             DATA   Word  %111111111111

EndOfPgms   DATA   0

Pgm0Len     CON     Wash - Zig
Pgm1Len     CON     EndOfPgms - Wash
```

Note the constant definitions that follow the DATA statements. These work because when the program gets compiled the symbols Zig, Wash, and EndOfPgms are actually converted to numeric values (the address of that location in EEPROM). With a little math we're able to determine the length of the sequences so that our program knows when to start over – and with this technique we can change the sequence length without any additional edits to our core code. Note, too, that we've used the Word modifier with DATA so each step is actually using two bytes of EEPROM; we'll need to account for this later.

And now for the main loop of the program. Its purpose is to read the step values from the selected sequence and send them to the designated outputs as we've already discussed.

```
Main:
DO
  LOOKUP PgmSelect, [Zig, Wash], baseAddr
  READ baseAddr + pgmStep, Word lightVal
  GOSUB Update_Outputs
  GOSUB Speed_Delay
  LOOKUP PgmSelect, [Pgm0Len, Pgm1Len], pgmMax
  pgmStep = pgmStep + 2 // pgmMax
LOOP
END
```

At the top of the loop we start by checking the sequence selection by using the value of PgmSelect (input P12) to control a LOOKUP table. If you want to extend to four sequences it's pretty simple: just add another switch to P13 and change that LOOKUP line to something like this:

```
LOOKUP (IND & %11), [Zig, Zag, Wash, Wear], baseAddr
```

The purpose of the LOOKUP line is to get the base (starting) address of the sequence that is currently selected. By doing this at the top of the loop we can change the sequence at any time. With the base address in hand, we read the selected DATA table for the current step value and move that into the variable lightVal, and then the target output structured is updated as we went through earlier.

The next line calls a subroutine called Speed_Delay that allows us to control the speed of the sequence with a potentiometer (see Figure 130.3 for the schematic). I get a lot of mail on this subject, so let me go through this in a little more detail so that you can add external speed controls to your programs.

Column #130: Lights, BASIC Stamp, Action!

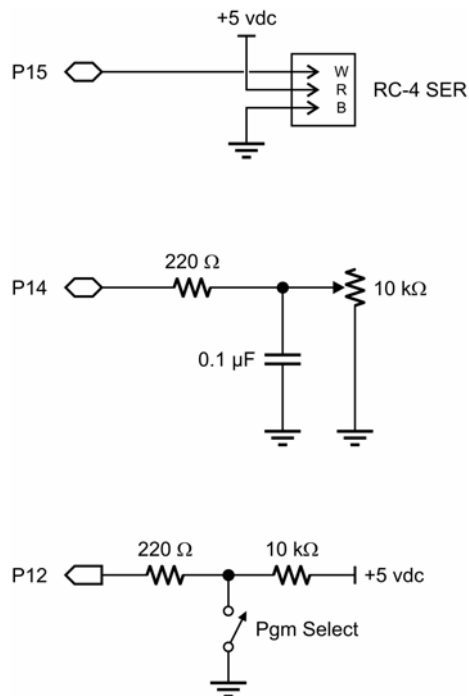


Figure 130.3: Lighting Control Schematic

While developing the program I started with this:

```
Speed_Delay:
HIGH Speed
PAUSE 1
RCTIME Speed, 1, delay
delay = delay * / Adjust
RETURN
```

This subroutine will read the pot with RCTIME and scale the value (using the conditional constant, Adjust) for the BASIC Stamp module in use. Remember, RCTIME is really just a type of stopwatch that measures the charge or discharge time of a capacitor. Each module has its own internal timing units so using Adjust lets us end up with the same return value, regardless of which BASIC Stamp module we put to use. The value of Adjust is set in a

#SELECT-#CASE structure that also sets the serial baud rate parameters for use with the RC-4 network.

Now we can use DEBUG to see what we get from the RCTIME circuit on the extreme ends of the potentiometer. On my system I got 18 on the low end, and 1124 on the high end. What we would like to do is rescale this range to 50 to 1000 milliseconds for our step delay. Here's the process:

The span of our raw input is 1106 (1124 minus 18). We divide this span into our desired output span of 950 (1000 minus 50) and end up with 0.858. You'll remember that we've used the ** operator in the past to multiply by fractional values of less than one, so that's what we'll do here. To convert 0.858 for use with ** we multiply the fraction by 65,536. The last part is to compensate for the low end of the output range (the "b" in the $mx + b$ equation). After the multiplication we add 35. Where did this come from? Since the low-end raw input is 18 and gets multiplied by 0.858 it will be reduced to 15. Our minimum new output value of 50 minus 15 is 35. DEBUG can be used to confirm our new range is very close to 50 to 1000. Knowing how we worked through the process, you can now readjust the program to create any minimum-to-maximum speed delay that you like.

An important lesson here is that there will be times when we have to work through a program empirically to get to the final result. Here's what we end up with:

```
Speed_Delay:
  HIGH Speed
  PAUSE 1
  RCTIME Speed, 1, delay
  delay = delay */ Adjust
  delay = delay ** $DBE4 + 35
  PAUSE delay
  RETURN
```

Getting back to the main code we're left with pointing to the next program step. Again, LOOKUP is used to determine which sequence is selected and assign the value of pgmMax (length of sequence in bytes). The next requirement is to add two to pgmStep – we have to do this because we're using Word-sized step values. The modulus operator takes care of wrapping the sequence back to its beginning.

And there we have it; a simple lighting control program that can send its output to a network of RC-4 relay boards or to Opto-22 SSRs using the Stamp CI board. I think this demonstrates the extraordinary flexibility of PBASIC2, especially when we use conditional compilation strategies. Okay, now how will you use this project to light up your Valentine's day?...

Column #130: Lights, BASIC Stamp, Action!

And just to show that we at Parallax actually put all this stuff to use, have a look at Figures 130.4 through 130.6. Figure 130.4 is Ryan Clarke's (Parallax Tech Support) RC-4 "Tower of Power" that he used to control a lighting sequence for his Christmas tree (it was a pretty fancy tree!). Figures 130.5 and 130.6 show a custom neon sign that my boss, Ken Gracey, and I built for trade shows. The neon part was, of course, contracted out to a sign shop. We took care of the control by using a BS2p on a Stamp CI board and a set of Opto-22 relays. If you happen to visit our Rocklin office, be sure to pop into the "Purple Room" for a look at the sign. It uses a pretty sophisticated program that includes IR remote control of the sequence and step speed.

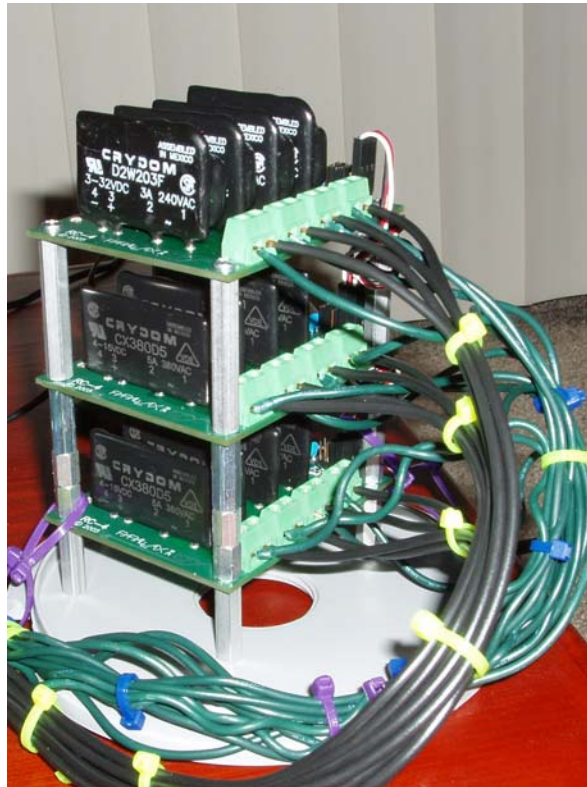


Figure 130.4: "Tower of Power"



Figure 130.5: The Parallax Neon Sign

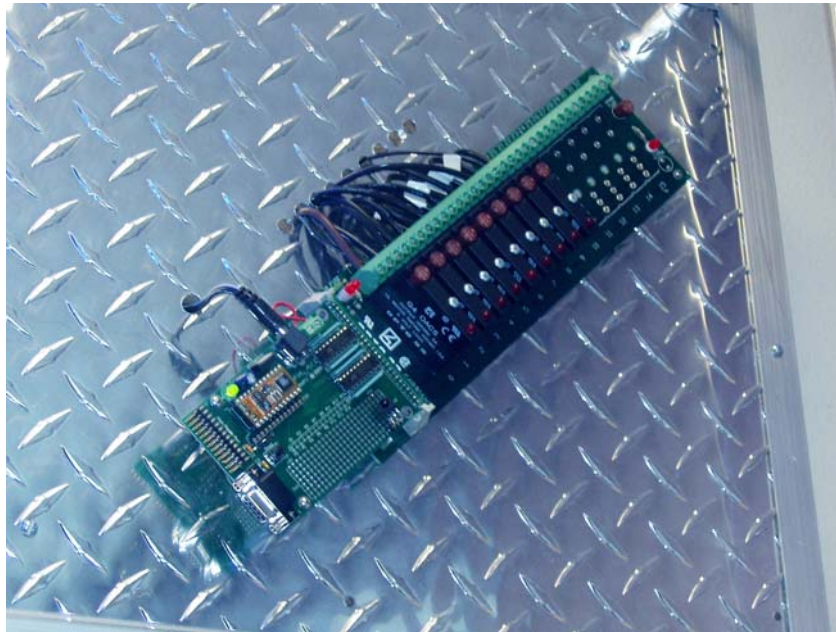


Figure 130.6: Stamp CI Board

Column #130: Lights, BASIC Stamp, Action!

One final note if you do use this program/circuit on the Stamp CI board (like we did with the neon sign): You need to cut the traces that connect P12-P14 to the output buffer. This will prevent the buffer circuitry from interfering with sequence selection and the RCTIME circuit.

Hacker's Hint

If you're into hacking store-bought products to make them your own, and you like weird and wacky items, you may want to be on the lookout for the various "talking" animals (fish, deer heads, etc.) that have become popular at large discount stores. The Parallax EFX gang (John Barrowman, Ryan Clarke, and me) bought a talking deer head (about \$100), gutted the electronics, and then installed a Prop-1 controller and an AP-8 audio player. It took about 15 minutes to determine which motor did what (all run at 9 volts DC, and we used the supply that came with the head to power the Prop-1 and AP-8). So, if you want to have fun with your friends and family, pop your own voice into an AP-8 and let your favorite animal deliver the message!

Happy Valentine's Day to all of you that are romantically inclined, and to all, Happy Stamping until we meet again next month! In April we'll be talking about a new Parallax controller that is so powerful it will make your head spin!

Project Code

```
' =====
'
'   File..... Lights.BS2
'   Purpose.... Light Control with RC-4 or Opto-22 relay board
'   Author..... Jon Williams -- Parallax, Inc.
'   E-mail..... jwilliams@parallax.com
'   Started....
'   Updated.... 15 DEC 2005
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =====

' -----[ Conditional Definitions ]-----

#DEFINE __SerialMode = 1                ' make 0 for parallel

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

#IF __SerialMode #THEN
  Sio          PIN      15              ' serial I/O for RC-4s
#ELSE
  Lights       VAR      OUTS           ' direct parallel outputs
#ENDIF
```

The Nuts and Volts of BASIC Stamps 2006

```

Speed          PIN      14          ' speed pot
PgmSelect     PIN      12          ' program selection

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

#SELECT $STAMP
#CASE BS2, BS2E, BS2PE
  T2400       CON      396
  T38K4       CON      6
  Adjust      CON      $200          ' RCTIME x 2 uS/unit
#CASE BS2SX
  T2400       CON      1021
  T38K4       CON      45
  Adjust      CON      $0CC          ' RCTIME x 0.8 uS/unit
#CASE BS2P
  T2400       CON      1021
  T38K4       CON      45
  Adjust      CON      $0C0          ' RCTIME x 0.75 uS/unit
#CASE BS2PX
  T2400       CON      1646
  T38K4       CON      84
  Adjust      CON      $0C0          ' RCTIME x 0.75 uS/unit
#ENDSELECT

SevenBit      CON      $2000
Inverted      CON      $4000
Open          CON      $8000

Baud          CON      Open + T38K4    ' open for RC-4 network

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

baseAddr      VAR      Word          ' base address of pgm
lightVal      VAR      Word          ' light outputs
pgmMax        VAR      Byte          ' length of program
pgmStep       VAR      Byte          ' current step
delay         VAR      Word          ' delay between steps

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

Zig           DATA    Word %000000000001
              DATA    Word %000000000010
              DATA    Word %000000000100
              DATA    Word %000000001000
              DATA    Word %000000010000
              DATA    Word %000000100000
              DATA    Word %000001000000
              DATA    Word %000010000000
              DATA    Word %000100000000
              DATA    Word %001000000000

```

Column #130: Lights, BASIC Stamp, Action!

```
DATA Word %010000000000
DATA Word %100000000000
Wash DATA Word %000000000000
DATA Word %000001100000
DATA Word %000011110000
DATA Word %000111111000
DATA Word %001111111100
DATA Word %011111111110
DATA Word %111111111111
EndOfPgms DATA 0
Pgm0Len CON Wash - Zig
Pgm1Len CON EndOfPgms - Wash

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

Reset:
#IF ( _SerialMode = 0 ) #THEN ' if parallel output mode
DIRS = $OFFF ' make P0-P11 outputs
#ENDIF
lightVal = $0000 ' clear lights
GOSUB Update_Outputs

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

Main:
DO
LOOKUP PgmSelect, [Zig, Wash], baseAddr ' base address of sequence
READ baseAddr + pgmStep, Word lightVal ' read step data
GOSUB Update_Outputs
GOSUB Speed_Delay
LOOKUP PgmSelect, [Pgm0Len, Pgm1Len], pgmMax
pgmStep = pgmStep + 2 // pgmMax
LOOP
END

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

' Causes a 50 ms to 1.0 second delay between sequence steps

Speed_Delay:
HIGH Speed ' charge RCTIME cap
PAUSE 1
RCTIME Speed, 1, delay ' read the speed pot
delay = delay * / Adjust ' adjust for module
delay = delay ** $DBE4 + 35 ' scale to 50 to 1000 mS
PAUSE delay
RETURN
```

```
' -----  
' Updates light channel outputs  
  
Update_Outputs:  
  #IF __SerialMode #THEN  
    SEROUT Sio, Baud, ["!RC4", %00, "S", lightVal.NIB0,  
                      "!RC4", %01, "S", lightVal.NIB1,  
                      "!RC4", %10, "S", lightVal.NIB2]  
  #ELSE  
    Lights = lightVal & $0FFF  
  #ENDIF  
  RETURN
```