Column #110 June 2004 by Jon Williams:

Drumming Up Control

Like most men, I'm not real big on the idea of shopping. I know what I want, I want what I want, and I know where to go get it – and that's precisely what I do: I go get it. Of course, for every rule there is an exception, and for my shopping rule there are two: book stores and Tanner Electronics in Dallas, Texas. I love going to Tanners, even when I don't need anything specific. Big Jim, Jimmy, Jake, Gina, and the rest of the family are really lovely people; they always have a smile for their customers and are helpful beyond the call of duty.

One of the many joys of going to Tanners is that I frequently run into BASIC Stamp users and am able to chat with them face-to-face. Back in late March I was in Tanners quite a lot as I was working on some prototypes for a new product. During my visits, I ran into quite a few BASIC Stamp users; and those collective meetings drove me to the project we're going to discuss this month.

The first of those customer meetings was with a man who was happy to see BS1 support in the BASIC Stamp Windows compiler and asked me to write more about the BS1, especially something that he could work through with his son. On another occasion that same week I chatted with two customers who have somewhat similar uses for the BASIC Stamp. One of those gentlemen is actually a medical doctor who's an avid Halloween enthusiast, and the

second works in the film industry as a special effects technician and prop builder. Both use the BASIC Stamp as a control element in their projects.

So what can we do with the BS1 that is simple enough for a young man, yet useful for the Halloween display builder and a professional special effects technician? We're going to build a drum sequencer. Now, before you jump on the Internet to Google "drum sequencer" let me warn you that you will be bombarded with hits having to do with Midi music controllers. That's not what we're doing.

Our drum sequencer is a simple controller that provides cyclical control of multiple outputs; that is, the controller will work through a series of control steps and upon reaching the end will restart the sequence at the beginning. There was a time when drum sequencers were in fact mechanical devices. A rotating drum with contact points or cams would provide control outputs to a number of circuits. If you're having a hard time visualizing this, just think of a mechanical music box. In a music box we turn a drum to play on or more notes and at the end of the song it starts over. A music box is a type of drum sequencer.

Traffic Control

Let's look at real-life example that is useful to explain the concept and provide a basis for developing our electronic version of this controller. Figure 110.1 shows a series of steps for elementary control of the traffic lights at an intersection. This is timed control only; there is no provision for external intelligence. The following code will illustrate a brute-force method of providing control.

```
Setup:
  DIRS = %00111111

Main:
  PINS = %00100001
  PAUSE 10000
  PINS = %00100010
  PAUSE 3000
  PINS = %00100100
  PAUSE 1000
  PINS = %00001100
  PAUSE 10000
  PINS = %00010100
  PAUSE 3000
  PINS = %00100100
  PAUSE 1000
  GOTO Main
```

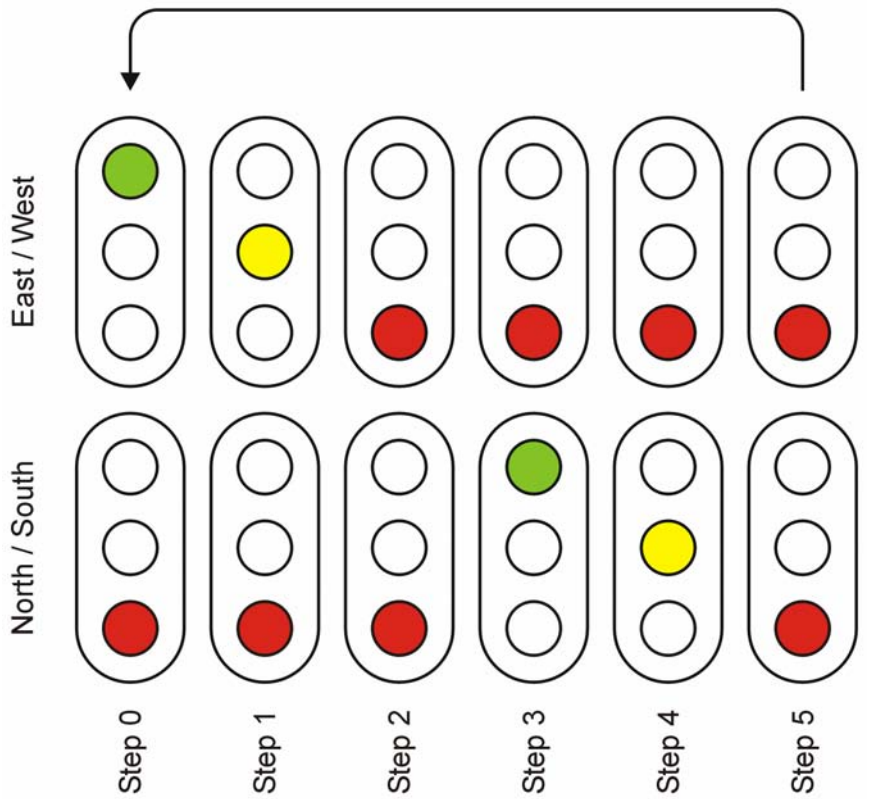

Figure 110.1: Typical Traffic Lights Sequence

If you connect LEDs to P0 – P5 (Figure 110.2) you'll see the sequence works just as we expect. That's great if it's all we want to do ... but we're human, and we know that somebody (maybe us) is going to ask for an adjustment or improvement. The first improvement we'll make is to move the output sequence and timing into a table. The reason for this is that we can adjust the elements of the sequence without digging into the heart of our control code.

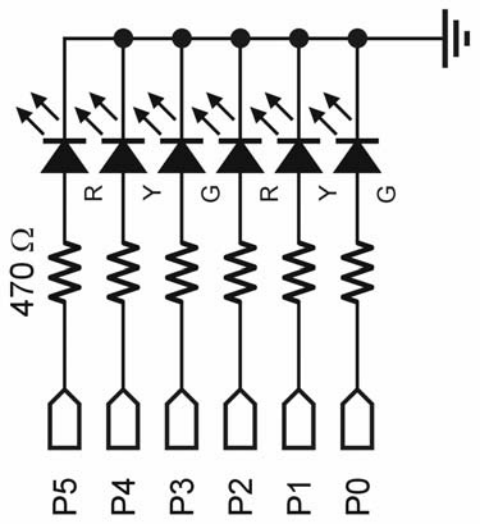

Figure 110.2: Schematic for Traffic Lights Simulation

Here's the code for our software-based drum:

```
Drum:
  EEPROM (%00100001, 10)
  EEPROM (%00100010,  3)
  EEPROM (%00100100,  1)
  EEPROM (%00001100, 10)
  EEPROM (%00010100,  3)
  EEPROM (%00100100,  1)
  EEPROM (%00100100,  0)
```

To make the table easy to read, each step is place on its own line and contains the outputs and timing value (in seconds). What you'll notice is that there is an extra step, and that the extra step has a timing value of zero. This isn't an operational step, of course, it's an indicator that we've reached the end of the table and it's time to start over. It's logical to use the time field as the end-of-table indicator as there will be other applications for the drum sequencer where all outputs are off.

Now that we've created a drum, let's create the code to "turn" it and activate the outputs:

```
Reset:
  DIRS = %00111111
  drumPntr = 0

Main:
  READ drumPntr, Lights
  drumPntr = drumPntr + 1
  READ drumPntr, stepTime
  drumPntr = drumPntr + 1
  IF stepTime = 0 THEN Reset
  timer = stepTime * StepUnits
  PAUSE timer
  GOTO Main
```

At the Reset section we start by enabling out output pins and setting the table pointer to zero. Normally, we don't need to initialize variables to zero as the BASIC Stamp does this for us, but in this program we will need to restart the sequence, so this is a convenient place to put that code.

At Main the real work gets done. The first READ from the table goes directly to the lights without the use of a holding variable – this takes advantage of the BASIC Stamp architecture and conserves a valuable resource. We're able to do this by creating the following definition:

```
SYMBOL  Lights          = PINS
```

After reading the control output, we must update the drum table pointer so that we can get the time for this step. READ is used again, and the pointer is updated one more time so that the next time through the loop it will be pointing at step outputs.

Just a moment ago we discussed using a zero time value and an indicator for the end-of-the table. So before bother timing the outputs, let's do that check. If the time value is zero, the program is routed back to Reset and the drum table pointer is reset to zero. By setting our end-ot-table outputs to the same as the previous step, there will be now glitch in the actual outputs as this check is taking place.

When we're not at the end (time value is greater than zero) we will multiply the time by a constant that will be useful for PAUSE. Here's where another design decision can make the program more flexible. Instead of embedding a "magic number" in the code, we declare a symbolic constant for milliseconds.

```
SYMBOL  StepUnits       = 1000
```

By doing this we can adjust the overall time of our sequence easily and by making just one edit. This also lets us run our sequence in an accelerated mode (by making StepUnits smaller) so that we can verify the sequence.

So now we've got a nice little traffic light simulator that we could use to sequence just about anything that will accept digital control outputs. For the moment, let's stick with the traffic light simulation and see how we can polish it up with the resources we have remaining.

The first thing that comes to my mind is that it would be nice to adjust the drum speed without having to crack open the code and edit it. If we were using an actual mechanical drum, we'd adjust its speed with a motor controller. We can simulate that in software quite easily.

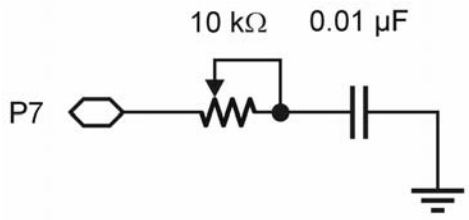

Figure 110.3: BS1 POT Input Circuit

By adding the circuit shown in Figure 110.3 we can read the value of a potentiometer using the BS1's POT function. The value returned by POT will range from zero to 255 – but let's add a little bit of design to our speed control. Let's say that we want it to scale the speed from 25% to 200% of normal. This is actually pretty simple: we'll scale the reading from POT so that it maximizes at 200 and will not go lower than 25.

```
Get_Speed:
  POT SpdInput, 103, scale
  scale = scale * 69 / 100 + 25
  RETURN
```

I'll bet you were expecting to use the MIN and MAX operators, weren't you? So why didn't we use them? The reason is that using MIN and MAX would have caused "dead" zones at either end of the pot's mechanical range – what we want is to use the entire range. Okay, then, how did we get there?

We start by taking the total span of our pot reading, 256 units, and dividing it into the span of our desired output which is 176 units (200 – 25 + 1). What we end up with is 0.686 that we can get very close to by multiplying by 69, then dividing by 100. After that we add in the minimum value of our output which is 25. Here's how the numbers work on the extreme ends of the pot's range:

```
0 x 69 = 0
0 / 100 = 0
0 + 25 = 25

255 x 69 = 17,595
17,595 / 100 = 175   (integer math)
175 + 25 = 200
```

Now that we have a scaling factor (25% to 200%) we can add it into the main loop of our program.

```
GOSUB Get_Speed
timer = stepTime * StepUnits / 100 * scale
```

If you're a little new to BASIC Stamp programming you may wonder why we're dividing by 100 before multiplying by our scale factor. Remember that the BASIC Stamp uses 16-bit integers and the largest value we can have is 65535 – anything greater will cause a roll-over error and lead to undesirable results. With the code the way it is, we can specify step times of up to 30 seconds (assuming StepUnits is 1000) without any timing problems.

Make It Special – Special Effects

By now I'm sure we've had enough fun with the traffic light simulation, so let's make some adjustments to our drum controller so that it better fits the needs of our friends who build cool Halloween attractions, or those that make movie props and special effects.

The first thing to do is expand the number of outputs, and it would be nice to do it in a manner that can be extended. No problem there, we'll use our old stand-by friend the 74HC595 shift register. Figure 110.4 shows the connections, and as you can see there is really nothing to it. The nice thing about the 74HC595 is that we can connect them in a daisy chain mode and get even more outputs (using pin 9 of one 74HC595 to feed the Din pin of the next). For the moment, let's just stick with one device for eight outputs.

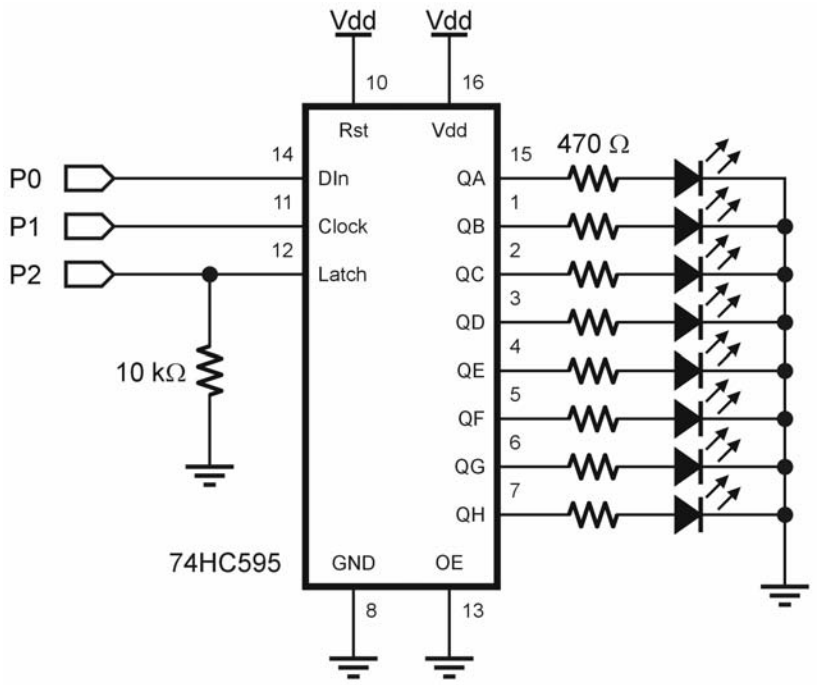

Figure 110.4: 74HC595 Connections

The 74HC595 is a synchronous serial device, but the BS1 doesn't have the SHIFTOUT instruction that is very popular with the BS2 family. Of course, this is not a big problem, we'll simply write a bit of code to take care of that function for us.

```
Shift_Out:
  FOR idx = 1 TO 8
    Dpin = dataMsb
    PULSOUT Clock, 1
    dOut = dOut * 2
  NEXT
  PULSOUT Latch, 1
  RETURN
```

This code is very simple, but there is a bit of hidden complexity that I want to reveal. This complexity has to do with the way that we declare variables for the BS1. You may have noticed in some of my other programs I start my variable definitions with B2, leaving B0 and B1 free. There is a very specific reason for this. The reason is that B0 and B1 are the only BS1 variables that are bit-addressable, so I make it a habit to reserve these in case I need to do something bit-oriented, as is required by the Shift_Out code.

With that, let's look at the variable declarations that allow the Shift_Out subroutine to work.

```
SYMBOL  dOut            = B0
SYMBOL  dataMSB         = BIT7
```

The variable called dOut is what we'll use to update the outputs (it will be destroyed in the process, so it's temporary). The variable dataMSB is BIT7 of the RAM memory map, and by our previous definition the MSB of dOut.

Getting back to the Shift_Out code we can see it's just a simple loop that will handle eight bits. That makes sense, right? Inside the loop the routine places the MSB of dOut onto the data pin of the 74HC595. Pulsing the clock line (low-high-low) causes the bit to be moved into the 74HC595. The next step is to shift the bits of dOut left so that we have a new MSB. This is done by multiplying dOut by two. After all the bits are moved into the 74HC595 we need to move the bits to the outputs. This is done by pulsing the Latch line.

Now, some of you may be thinking that with all that work it must take forever to update the outputs. It doesn't. I put the subroutine into a counter loop and found that even with the "old" BS1, the 74HC595 outputs can be updated about 20 times per second – much faster that we're ever going to need in this kind of application.

The next upgrade to our controller is going to a trick I learned a long time ago when working in the irrigation industry building sprinkler controllers. We're going to apply a bit of encoding to the time field so that we can have more flexibility on that end. In or original design – with everything set at 100% – we can get step durations up to 25 seconds. Maybe we have a holiday lighting display that needs a step output longer than this.

Here's the trick: We're going to use bit 7 of the time field as a multiplier (something other than two as it is now, that is). In this version of the program, bit 7 will tell the program to multiply the time value by 10, so now we have two ranges: 0.1 to 12.7 seconds per step in 0.1 second increments, and 1 to 127 seconds in one second increments. Take a look at this drum table:

```
Drum:
  EEPROM (%10000001, %00000101)
  EEPROM (%01000010, %00001010)
  EEPROM (%00100100, %10000010)
  EEPROM (%00011000, %10000011)
  EEPROM (%00000000, 0)
```

The first two steps have bit 7 of the time field clear, so these steps will run in increments of 0.1 seconds, in this case step 0 will run for 0.5 seconds and step 1 will run for one second. Steps 2 and 3 have bit 7 set, so the multiplier (x10) will be used. This means that step 2 will run for two seconds, and step 3 will run for three seconds. Let's have a look at the code that takes care of this:

```
Step_Timer:
  timer = stepTime & %0111111
  IF longStep = No THEN Timer_Loop
  timer = timer * Multiplier

Timer_Loop:
  POT SpdInput, 103, delay
  delay = delay * 69 / 100 + 25
  FOR idx = 1 TO timer
    PAUSE delay
  NEXT
  RETURN
```

To make things convenient the timing code has been moved into a subroutine. At the top of this code we'll move the base step time to timer – without the multiplier bit. Then we can check the multiplier bit (aliased as longStep) and if it is clear (0) we will jump right to the timing loop, otherwise we will apply the multiplier to our base time.

After that we move into the timing section which starts by reading the pot input. To keep things clean, the potentiometer is read and provides direct timing instead of an adjustment factor as we did in the final version of the traffic light simulation. The last step is to use timer as a loop control to create the step timing. The advantage of this method is that the maximum PAUSE duration will be 200 milliseconds (read and scaled from the potentiometer input), so if we want to interrupt the cycle with some sort of override, we don't have to wait for the entire step to time out. By inserting an "escape route" in the final timing loop we will only ever have to wait 200 milliseconds to interrupt a step.

All right, let's just add one more feature to our controller for before wrapping up, shall we? Since we're now using the 74HC595 to handle the outputs, we have some free IO pins on the

BS1. It seemed to me that the ability to add a single-step mode to the controller would be useful. This could be used to check the outputs sequence, or for manual control when step timing is variable (as when operating a prop or film effect).

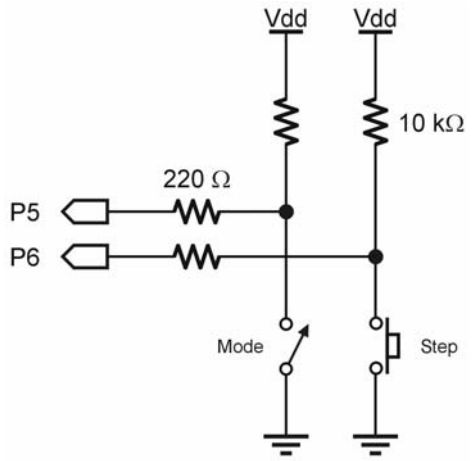

Figure 110.5: Mode Switch and Step Inputs

By using the switch and pushbutton inputs shown in Figure 110.5, we can add the single-step feature to the controller. The switch will serve as our Mode input: open (P5 will be pulled high) means that we're going to run using timed steps, closed (P5 will be pulled low) means that the Step button will be used to manually advance the sequence.

```
Main:
  READ drumPntr, stepOuts
  drumPntr = drumPntr + 1
  READ drumPntr, stepTime
  drumPntr = drumPntr + 1
  IF stepTime = 0 THEN Reset
  dOut = stepOuts
  GOSUB Shift_Out
```

```
Check_Mode:
  IF Mode = MdTimer THEN Timed_Step

Force_Release:
  IF Advance = Pressed THEN Force_Release

Wait_For_Press:
  IF Advance = NotPressed THEN Wait_For_Press
  GOTO Main

Timed_Step:
  GOSUB Step_Timer
  GOTO Main
```

Even adding the timed versus single-step option doesn't complicate our program at all. After updating the outputs with the current step data, the program checks the state of the mode switch. If open, it jumps to Timed_Step which, in turn, calls the Step_Timer subroutine (remember that PBASIC in the BS1 is very close to assembly language in many respects, so there is no IF-THEN-ELSE and we can only branch with IF-THEN).

When the Mode switch is closed (single-step mode selected) the program will examine the state of the step button (the input is called Advance in the program). The first thing it does is make sure that it's not already pressed. This was a design decision I made so that each step requires a separate button press and release. You can eliminate this if you like, but you may find a small PAUSE is required between steps, because without it a simple button press may result in the execution of several steps.

Once we know the button is clear we wait for it to close, and when it does the program branches back to Main where we execute the next step. Pretty easy, isn't it? – yet a very cool and useful project for many applications. Another option we could add to the program is the ability to single-shot a timed sequence. But, alas, we're out of space so I will leave that to you. Of course, if you need to do that and can't figure it out you can always send me a note.

Have fun, and until next time – Happy Stamping.

```
' ================================================================================
'
'   File..... Traffic_Lights_(Brute).BS1
'   Purpose... Simple Traffic Light Demo
'   Author.... Jon Williams
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 11 APR 2004
'
'   {$STAMP BS1}
'   {$PBASIC 1.0}
'
' ================================================================================


' -----[ Program Description ]--------------------------------------------


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


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


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


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


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


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

Setup:
  DIRS = %00111111


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

Main:
  PINS = %00100001
  PAUSE 10000
  PINS = %00100010
  PAUSE 3000
  PINS = %00100100
  PAUSE 1000
  PINS = %00001100
```

```
  PAUSE 10000
  PINS = %00010100
  PAUSE 3000
  PINS = %00100100
  PAUSE 1000
  GOTO Main

  END


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

```
' ================================================================================
'
'   File..... Traffic_Lights_(Drum).BS1
'   Purpose... Simple Traffic Light Demo
'   Author.... Jon Williams
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 11 APR 2004
'
'   {$STAMP BS1}
'   {$PBASIC 1.0}
'
' ================================================================================


' -----[ Program Description ]--------------------------------------------


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


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

SYMBOL  Lights          = PINS


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

SYMBOL  StepUnits       = 1000                  ' for PAUSE, 1 second


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

SYMBOL  drumPntr        = B2                    ' drum pointer
SYMBOL  stepTime        = B3                    ' step time in seconds
SYMBOL  timer           = W2                    ' timer value for PAUSE


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

Drum:
  EEPROM (%00100001, 10)
  EEPROM (%00100010,  3)
  EEPROM (%00100100,  1)
  EEPROM (%00001100, 10)
  EEPROM (%00010001,  3)
  EEPROM (%00100100,  1)
  EEPROM (%00100100,  0)


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

```
Reset:
  DIRS = %00111111
  drumPntr = 0


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

Main:
  READ drumPntr, Lights
  drumPntr = drumPntr + 1
  READ drumPntr, stepTime
  drumPntr = drumPntr + 1
  IF stepTime = 0 THEN Reset
  timer = stepTime * StepUnits
  PAUSE timer
  GOTO Main

  END


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

```
' ================================================================================
'
'   File..... Traffic_Lights_(Final).BS1
'   Purpose... Simple Traffic Light Demo
'   Author.... Jon Williams
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 11 APR 2004
'
'   {$STAMP BS1}
'   {$PBASIC 1.0}
'
' ================================================================================


' -----[ Program Description ]---------------------------------------------


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


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

SYMBOL  SpdInput        = 7                     ' speed controller input
SYMBOL  Lights          = PINS                  ' output to lights


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

SYMBOL  StepUnits       = 1000                  ' for PAUSE, 1 second


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

SYMBOL  drumPntr        = B2                    ' drum pointer
SYMBOL  stepTime        = B3                    ' step time in seconds
SYMBOL  scale           = B4                    ' speed scale factor
SYMBOL  timer           = W3                    ' timer value for PAUSE


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

Drum:
  EEPROM (%00100001, 10)                        ' step outs, seconds
  EEPROM (%00100010,  3)
  EEPROM (%00100100,  1)
  EEPROM (%00001100, 10)
  EEPROM (%00010100,  3)
  EEPROM (%00100100,  1)
  EEPROM (%00100100,  0)
```

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

Reset:
  DIRS = %00111111
  drumPntr = 0


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

Main:
  READ drumPntr, Lights
  drumPntr = drumPntr + 1
  READ drumPntr, stepTime
  drumPntr = drumPntr + 1
  IF stepTime = 0 THEN Reset
  GOSUB Get_Speed
  timer = stepTime * StepUnits / 100 * scale
  PAUSE timer
  GOTO Main

  END


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

Get_Speed:
  POT SpdInput, 103, scale                      ' read speed pot
  scale = scale * 69 / 100 + 25                 ' scale to 25 - 200
  RETURN
```

```
' ================================================================================
'
'   File...... Drum_Sequencer.BS1
'   Purpose... General-purpose drum sequencer
'   Author.... Jon Williams
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 11 APR 2004
'
'   {$STAMP BS1}
'   {$PBASIC 1.0}
'
' ================================================================================


' -----[ Program Description ]----------------------------------------------


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


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

SYMBOL  Dpin            = PIN0                  ' to 74HC595.14
SYMBOL  Clock           = 1                     ' to 74HC595.11
SYMBOL  Latch           = 2                     ' to 74HC595.12
SYMBOL  Mode            = PIN5                  ' timed or step
SYMBOL  Advance         = PIN6                  ' manual step advance
SYMBOL  SpdInput        = 7                     ' speed controller input


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

SYMBOL  Yes             = 1                     ' use multiplier
SYMBOL  No              = 0

SYMBOL  MdTimer         = 1                     ' timed steps
SYMBOL  MdStep          = 0                     ' single step

SYMBOL  Pressed         = 0                     ' for active low button
SYMBOL  NotPressed      = 1

SYMBOL  Multiplier      = 10                    ' timing multiplier


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

SYMBOL  dOut            = B0                    ' data for 74HC595
SYMBOL  stepTime        = B1                    ' step timing (compressed)
SYMBOL  stepOuts        = B2                    ' step outputs
SYMBOL  drumPntr        = B3                    ' drum pointer
```

```
SYMBOL  delay           = B4                    ' inner loop timing delay
SYMBOL  idx             = W3                    ' loop counter
SYMBOL  timer           = W4                    ' step timer

SYMBOL  dataMSB         = BIT7                  ' MSB of step data
SYMBOL  longStep        = BIT15                 ' Bit7 or stepTime


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

Drum:
  EEPROM (%10000001, %00000101)
  EEPROM (%01000010, %00001010)
  EEPROM (%00100100, %10000010)
  EEPROM (%00011000, %10000011)
  EEPROM (%00000000, 0)                         ' end of sequence


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

Reset:
  PINS = %00000000
  DIRS = %00000111
  drumPntr = 0                                  ' reset drum


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

Main:
  READ drumPntr, stepOuts                       ' get outputs from drum
  drumPntr = drumPntr + 1                       ' point to step time
  READ drumPntr, stepTime                       ' get step time from drum
  drumPntr = drumPntr + 1                       ' point to next step
  IF stepTime = 0 THEN Reset                    ' 0 time means start over
  dOut = stepOuts
  GOSUB Shift_Out                               ' update outputs

Check_Mode:
  IF Mode = MdTimer THEN Timed_Step             ' check mode

Force_Release:
  IF Advance = Pressed THEN Force_Release

Wait_For_Press:
  IF Advance = NotPressed THEN Wait_For_Press
  GOTO Main

Timed_Step:
  GOSUB Step_Timer
  GOTO Main
```

```
  END


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

Shift_Out:
  FOR idx = 1 TO 8                              ' shift eight bits
    Dpin = dataMsb                              ' put MSB on data pin
    PULSOUT Clock, 1                            ' clock it out
    dOut = dOut * 2                             ' shift data bits left
  NEXT
  PULSOUT Latch, 1                              ' latch new byte to outputs
  RETURN


Step_Timer:
  timer = stepTime & %0111111                   ' get base time
  IF longStep = No THEN Timer_Loop              ' check multiplier bit
  timer = timer * Multiplier                    ' -- use mutliplier

Timer_Loop:                                     ' run the delay
  POT SpdInput, 103, delay                      ' read speed pot
  delay = delay * 69 / 100 + 25                 ' scale to 25 - 200
  FOR idx = 1 TO timer
    PAUSE delay
  NEXT
  RETURN
```