Column #135, July 2006 by Jon Williams:

A Tale of Two Props (Advanced BS1 Programming)

After spending a few months with Parallax's newest controller (the Propeller chip), why not step back in time a bit (13 years!) and work with the oldest, the venerable BS1 – the controller that started it all for Parallax. I recently noted that these days (with the Propeller) remind me of the first months of the BS, when everything was new, the rules had changed, what wasn't possible yesterday suddenly was today. Of course, the BS1 has found new life as the core of Prop-1 controller, and despite its low cost, many customers have asked to "more," that is, some actually want to control two props – independently – with one controller. Can we do it? Sure, but it will test our BS1 programming skills.

Here it is, the middle of summer, it's hot as Hades in many places and we're already talking about Halloween prop control. The truth is that real die-hards have been at it for a couple months now, and Parallax has attended and participated in several regional trade shows that attract Halloween prop builders. In fact, you can catch us this month in Columbus, Ohio, at the Midwest Haunters Convention. No doubt there will be lots of discussion on prop programming – we're actually doing a short seminar on the subject.

When we designed the Prop-1 after HauntX in 2005 we set the price at a point where we felt users could dedicate one controller per prop. Did that happen? Well, mostly, but there are those on a real serious budget that turned to us for guidance – they expressed a need to control two completely-independent props with just a single Prop-1 controller.

One of our customers, a guy named Allan, had a bit of success, which was quite impressive for someone who hadn't spent a great deal of time with the BS1 – remember, PBASIC1 is not nearly as fancy as other flavors of BASIC. He and I exchanged ideas and it finally occurred to me that I could use tricks developed for a commercial product (BS2-based) to create a BS1-based multi-prop controller program.

The KISS Prop Principal

Most of the prop builders we work with are not hardcore programmers and really don't care to be, so they tend to stick with the KISS prop principal:

A. Activate an output
B. PAUSE as required by the output
C. Deactivate the output

This works perfectly well for single-prop controls, but when we get into multiple props on one controller we're stepping it up a bit; in essence, we're creating and running multiple "threads" of program code. Of course, the Prop-1 is a very simple controller, and I'm not suggesting for a millisecond that it's "multi-threaded." That said, we can employ techniques in high-level PBASIC that mimic "time-slicing" in more sophisticated systems.

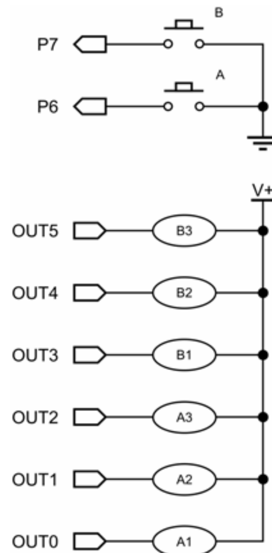

Figure 135.1: Dual Controller Connections

Dual Action Demolition

Controlling two sets of outputs simultaneously requires a different approach – we certainly can't allow one prop to be hung-up on a PAUSE instruction while the other prop needs to change the output states of its I/O pins. So, rule number one is that PAUSE – for event timing – is out.

How, then, do we control the timing of the outputs? Some of you may remember a program we did a few years ago called a "drum sequencer." This coding technique is the software analog a player piano; as the "sheet" is scrolled, events are picked up and "played."

We can create a play "sheet" (outputs sequence) in an EEPROM table, and include the timing between events to control how quickly things move. The timing will actually control the duration between table reads, therefore controlling the timing between possible output changes. So, rule number two of a dual prop controller on the BS1 is that events are table-driven.

Okay, how are we going to get some known timing element into the prop control? The answer is deceptively simple: use the trigger input check code to create a specific delay. In order to keep things really simple, I use 100 ms. Let's have a look:

```
Main:
  FOR loopTmr = 1 TO 5
    seq1 = seq1 | Trigger1
    seq2 = seq2 | Trigger2
    PAUSE 20
  NEXT
```

As you can see, this looks a bit different than our stock trigger-detection loop – usually such a loop waits until a trigger is present before dropping through. In this case, though, what the loop does is update (if a trigger input is present) one or both bit status flags (seq1 and seq2) for the sequences. The loop always runs five times, and with a 20 ms PAUSE embedded in the loop it takes just a bit longer than 100 ms to scan the inputs. This 100 ms delay will be used as our core timing element (we're going to keep things simple and ignore program overhead).

Let me take a small detour for a moment on a BS1 programming habit. When I write code for the BS1 I always start assigning variables at B2, leaving B0 and B1 available for future use. Why? Because these are the only two BS1 variables that allow bit-level access, and in this program we need them. Since most BS1 programs are small and we generally have enough RAM, start your assignments at B2 so that you have the bits in B0 and B1 available if an update requires them.

Back to the program: note that the OR operator is used in the trigger loop. The reason for this is that seq1 and seq2 denote the running state (1 is running, 0 is not running) of the respective sequences. By using the OR operator we can start a sequence (0 OR 1 = 1) and if the sequence is already running, there is no change on a subsequent trigger input (1 OR 1 = 1). Note that the Prop-1 inputs pins, P6 and P7, have been set to active-high, that is, their SETUP jumpers have been moved to the DN (pull-down) position.

Table Manners

Now things get a little bit heavy; our code has to slice-and-dice two sequence tables, dealing with each step in each table as independent entities. In the simplest terms, our program is doing this:

1. Timing/Trigger input loop
2. Process sequence 1 if running
3. Process sequence 2 if running

Steps 2 and 3 above are identical; the only difference being the pointer to the tables used and the escape destinations (Sequence 1 escapes to Sequence 2, Sequence 2 escapes back to Main). Since we're talking about tables, let's have a look them as they're structured in the program:

```
SYMBOL  Seq1Start       = 0
SYMBOL  Seq1Mask        = %00111000

Sequence1:
  EEPROM (%00000001, 5)
  EEPROM (%00000010, 5)
  EEPROM (%00000100, 5)
  EEPROM (%00000010, 5)
  EEPROM (%00000001, 5)
  EEPROM (%10000000, 0)

SYMBOL  Seq2Start       = 12
SYMBOL  Seq2Mask        = %00000111

Sequence2:
  EEPROM (%00100000, 2)
  EEPROM (%00010000, 2)
  EEPROM (%00001000, 2)
  EEPROM (%10000000, 0)
```

Notice, too, that we're embedding some named constants in the table section; we don't normally do this, but for BS1-based table programs it's a good idea as we have to assign these

values manually; keeping the assignment code close is just handy, especially after changing the length of either of the tables.

For each sequence we have a start value: this is the location in EEPROM where the actual table data starts. The second constant is a pins mask for the other sequence; this will be used for clearing the pins of one sequence without disturbing the pins under control by the other. Remember, this code is supposed to behave like two independent sequencers, so I/O pins are assigned to just one sequence and cannot be controlled by both.

Finally, each table entry consists of two bytes: the first is the outputs (with an end-of-sequence flag embedded in bit 7), and the second is the timing value, which is expressed in units of 100 ms (from the timing input loop).

Now for the nitty-gritty; let's have a look at how the sequence processing works. This part of the program is divided into two sections:

A. If sequence running and timer active, decrement timer
B. Load next step (outputs and timer)

Section A is labeled Run_X (X is 1 or 2) in the program. Here's the code:

```
Run_1:
  IF seq1 = Stopped THEN Run_2
    IF timer1 = 0 THEN Reload_1
      timer1 = timer1 - 1
      GOTO Run_2
```

Remember that with the BS1 we don't have IF-THEN-ELSE, we're limited to IF-THEN-Address; this is not a programming problem, it just means that we have to be very deliberate in our design, often using what seems to be inverted logic. In this case we want to update the timer if the sequence is running (seq1 = 1), so our first check to see if it's stopped (seq1 = 0); if the sequence is stopped then we blow right by the rest of this section and deal with sequence 2.

Let's say that the sequence is running and that we have time left for the current table entry; what we do is decrement the sequence timer value and then move on to the next sequence. Again, the timing delay is created at the trigger input loop so there is no need to put a PAUSE here.

When the sequence is running and the timer has expired we will jump to the next part of the sequence processor: Reload_1.

```
Reload_1:
  READ pntr1, pinsTemp
  PINS = PINS & Seq1Mask | pinsTemp
  pntr1 = pntr1 + 1
  READ pntr1, timer1
  pntr1 = pntr1 + 1
  IF endOfSeq = No THEN Run_2
    seq1 = Stopped
    pntr1 = Seq1Start
```

At Reload_1 we begin by reading the next table entry into pinsTemp. This is then sent to the outputs after ANDing the current pins state with the protection mask. Just for clarification, the mask for sequence 1 is designed to protect the pins for sequence 2 (and vice-versa). With the pins updated, the record pointer is incremented to point to the timing for this step. This gets read into timer1 and the pointer is incremented a second time to point to the next table entry.

If this is not the end of the sequence the program will make its way back to the top and the timer will be updated in the Run_1 section. If you look closely at the last entry in each table you'll see that the pins output value has bit 7 set; this is used as the end-of-sequence indicator. Since pinsTemp is assigned to B1, the variable endOfSeq is aligned in BIT15 (bit7 of B1, which we've aliased to make the program easier to follow).

If endOfSeq is 1 the program will clear the status variable, seq1, and then reset the table pointer, pntr1, back to the beginning. The code for sequence 2 is identical, the only difference being that it has jumps back to the top of the program at Main.

And there you have it – on a tiny little Prop-1 controller we have the ability to control two props, completely independently of each other. And you thought it took a big PC to do time-slicing! Okay, I'm exaggerating a bit, but that is in essence what we're doing.

Even Farther?

Believe it or not, we can go even further with this technique, but we'll save that for another day. I have actually written a program that will control two sequences and can use the Prop-1 outputs as well as an RC-4 for AC outputs (lights, etc.). The program works, but it's a bit of a bear, and with all the code to manage parallel and RC-4 outputs it doesn't leave a lot of room left for sequence tables.

Okay, gang, Halloween is not that many months away so start your code writing now – and if you're in a budget pinch you can double-up your Prop-1. And if you're in the Columbus area

this month, do come by the Midwest Haunters Convention and say hello to John Barrowman and me. We'll have a booth there and love to meet new friends.

Until next time, Happy Stamping.

Resources:

Midwest Haunters Convention
http://www.midwesthauntersconvention.com

Project Code

```
' =========================================================================
'
'   File....... Dual-Sequencer.BS1
'   Purpose....
'   Author..... Jon Williams -- Parallax EFX
'               -- based on code by Allen Huffman
'   E-mail..... jwilliams@parallax.com
'   Started....
'   Updated.... 17 MAY 2006
'
'   {$STAMP BS1}
'   {$PBASIC 1.0}
'
' =========================================================================

' -----[ Program Description ]---------------------------------------------
'
' This program converts the Prop-1 into a dual timer capable of simultane-
' ously and independently controlling two individual props.  The outputs
' for each prop are defined by an EEPROM table.  Each record in the table
' holds the new outputs for a given state as well as the timing for that
' state.  Bit7 of the outputs value indicates the end of the sequence. It
' is the programmer's responsibility to set Seq1Start and Seq2Start to
' the proper values before downloading the program.
'
' P6 and P7 are setup for active-high inputs -- move the SETUP jumpers to
' the DN position.

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

SYMBOL  Trigger2        = PIN7                  ' move setup to DN
SYMBOL  Trigger1        = PIN6                  ' move setup to DN

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

SYMBOL  IsOn            = 1                     ' for active-high input
SYMBOL  IsOff           = 0
```

```
SYMBOL  Running         = 1                     ' for status flags
SYMBOL  Stopped         = 0

SYMBOL  Yes             = 1
SYMBOL  No              = 0

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

SYMBOL  status          = B0                    ' sequence status flags
SYMBOL  seq1            = BIT0
SYMBOL  seq2            = BIT1
SYMBOL  pinsTemp        = B1                    ' temporary output
SYMBOL  endOfSeq        = BIT15                 ' 1 = end of sequence
SYMBOL  loopTmr         = B2
SYMBOL  pntr1           = B3                    ' pointer for seq 1
SYMBOL  timer1          = B4                    ' timer for seq 1
SYMBOL  pntr2           = B5
SYMBOL  timer2          = B6

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

' Timing value is in ~100 millisecond units
' -- trigger check loop is 5 iterations x 20 ms

SYMBOL  Seq1Start       = 0                     ' start point in EE
SYMBOL  Seq1Mask        = %00111000             ' protect P3-P5

Sequence1:
  EEPROM (%00000001, 5)
  EEPROM (%00000010, 5)
  EEPROM (%00000100, 5)
  EEPROM (%00000010, 5)
  EEPROM (%00000001, 5)
  EEPROM (%10000000, 0)                         ' end of sequence

SYMBOL  Seq2Start       = 12                    ' start point in EE
SYMBOL  Seq2Mask        = %00000111             ' protect P0-P2

Sequence2:
  EEPROM (%00100000, 2)
  EEPROM (%00010000, 2)
  EEPROM (%00001000, 2)
  EEPROM (%00100000, 2)
  EEPROM (%00010000, 2)
  EEPROM (%00001000, 2)
  EEPROM (%00100000, 2)
  EEPROM (%00010000, 2)
  EEPROM (%00001000, 2)
  EEPROM (%00100000, 2)
  EEPROM (%00010000, 2)
  EEPROM (%00001000, 2)
```

```
  EEPROM (%00100000, 2)
  EEPROM (%00010000, 2)
  EEPROM (%00001000, 2)
  EEPROM (%10000000, 0)

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

Reset:
  PINS = %00000000                                ' clear outputs
  DIRS = %00111111                                ' set to output mode

  pntr1 = Seq1Start                               ' initialize pointers
  pntr2 = Seq2Start

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

Main:                                             ' test triggers, pad loop
  FOR loopTmr = 1 TO 5                            ' text 5x
    seq1 = seq1 | Trigger1                        ' update running flag
    seq2 = seq2 | Trigger2                        ' update running flag
    PAUSE 20                                      ' 5 x 20 ms = ~100 ms
  NEXT

Run_1:
  IF seq1 = Stopped THEN Run_2                    ' running?
    IF timer1 = 0 THEN Reload_1                   '  yes, timer expired?
      timer1 = timer1 - 1                         '   no, decrement timer
      GOTO Run_2

Reload_1:
  READ pntr1, pinsTemp                            ' read output state
  PINS = PINS & Seq1Mask | pinsTemp               ' apply new outputs
  pntr1 = pntr1 + 1                               ' point to timing
  READ pntr1, timer1                              ' read it
  pntr1 = pntr1 + 1                               ' point to next record
  IF endOfSeq = No THEN Run_2                     ' at end?
    seq1 = Stopped                                ' - stop sequence
    pntr1 = Seq1Start                             ' - reset pointer

Run_2:
  IF seq2 = Stopped THEN Main                     ' running?
    IF timer2 = 0 THEN Reload_2                   '  yes, timer expired?
      timer2 = timer2 - 1                         '   no, decrement timer
      GOTO Main

Reload_2:
  READ pntr2, pinsTemp                            ' read output state
  PINS = PINS & Seq2Mask | pinsTemp               ' apply new outputs
  pntr2 = pntr2 + 1                               ' point to timing
  READ pntr2, timer2                              ' read it
  pntr2 = pntr2 + 1                               ' point to next record
```

```
IF endOfSeq = No THEN Main                        ' at end?
  seq2 = Stopped                                  ' - stop sequence
  pntr2 = Seq2Start                               ' - reset pointer

GOTO Main
```