Column #141, January 2007 by Jon Williams:

Ready...Set...Code!

Having been a part of the BASIC Stamp community since 1994, I've had the wonderfully-good fortune to meet a lot of experimenters, and – due to my exposure through Nuts & Volts and six years with Parallax – I've been asked to create a wide variety of projects. One of the most frequently requested, but that is not really practical with a BASIC Stamp, is a Pinewood Derby racing timer. Well, now that programming the SX is nearly as easy as programming the BASIC Stamp, the race timer can finally be realized – and even provide one-millisecond resolution using nothing but BASIC.

If you've read my past articles on SX/B you may remember that I've always taken a bit of a cautionary position when it comes to using interrupts. We must always keep in mind that interrupts steal time from the foreground program, and we must be particularly mindful when using time-sensitive instructions like PAUSE, SEROUT, etc. That said, there are times when using interrupt-driven code is actually the better choice over linear programming. Our race timer is one of those kinds of projects; especially since we want to create a timer with a one-millisecond resolution while multiplexing a multi-digit LED display.

For a compiler that's absolutely free, SX/B does a fantastic job with interrupts – better than most BASIC compilers that cost hundreds of dollars. Still, interrupts should be approached carefully. With a bit of thought and planning we can have interrupts without headaches running on the SX in SX/B. In fact, we'll see that with the improvements since SX/B 1.51, this project becomes nearly trivial.

First things first: what does the race timer need to do? For starters (no pun intended) the timer should accept a remote start signal so that it can be used in single- or multi-track setups. We'll also need a remote clear signal so that we can reset the time to zero before a race. When running we need to keep track of the time and, most importantly, display it on a five-digit, seven segment display.

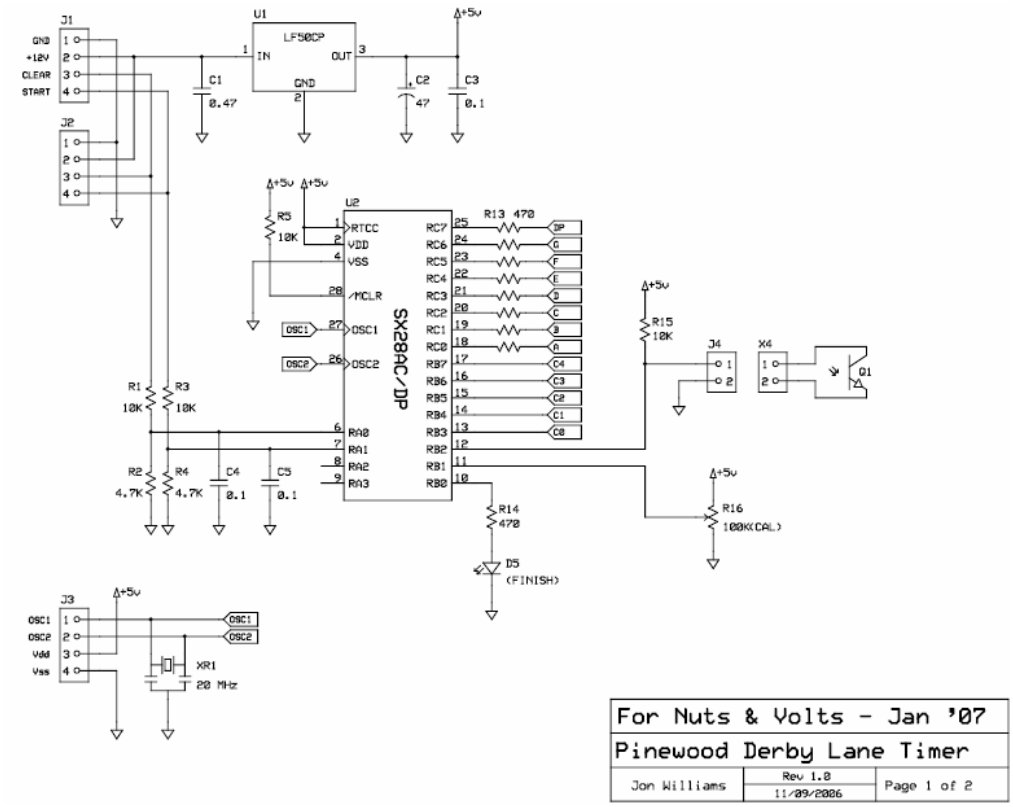

Figure 141.1: Pinewood Derby Lane Timer, Page 1 of Schematic

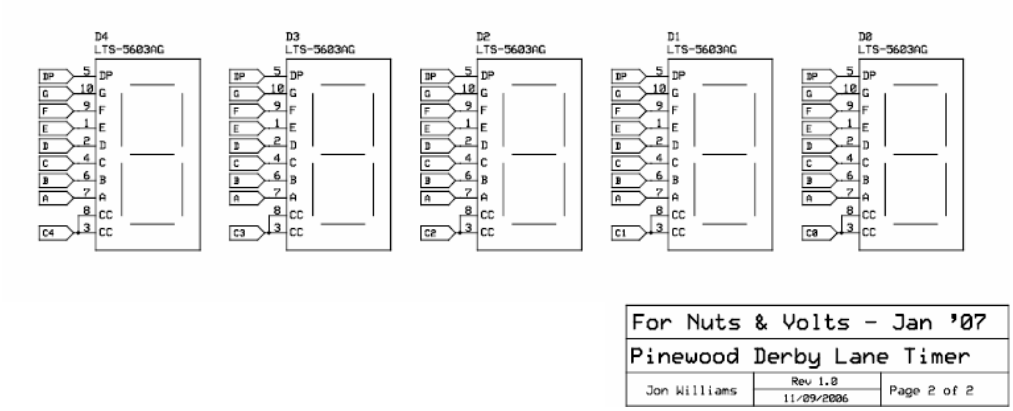

Figure 141.2: Pinewood Derby Lane Timer, Page 2 of Schematic

Figures 141.1 and 141.2 show the schematic for the timer using an SX28. Much of the I/O is devoted to the 7-segment displays, with a few bits to monitor the finish line photo-transistor and Clear and Start inputs. Note that power and control inputs come in and go back out on 4-pin connectors. The idea is that this will make construction of multi-track setups simpler. Since the power connection provides 12 volts the Clear and Start signals are also 12 volts. A two-resistor divider brings the signal voltage down to about 3.8 volts; this is well above the TTL switching threshold of 1.4 volts for the SX. Note, too, that we're using the SX comparator input to detect finish. By using the comparator we can adjust the input set-point (with R16) and accommodate a wide variety of ambient lighting conditions.

From the firmware standpoint, the two most critical elements of the program are the event timing and display multiplexing – this is where an interrupt will come in handy. How about if we interrupt the program every millisecond to update the timer and the display – that's easy, right? Yes, it absolutely is.

Periodic interrupts in the SX are controlled by the OPTION register, and when using assembly or SX/B versions prior to 1.51 we would have to set this register manually. This is not a huge hassle, but if we decided to change the clock frequency or interrupt rate we'd have to do it again. Well, not any more. Since we want to interrupt the program every millisecond – or 1000 times per second – we simply tell the SX/B compiler that's what we want to do:

```
INTERRUPT 1000
```

Yes, that's it. As of SX/B version 1.51 the compiler will calculate proper value for the OPTION register and will put it into the startup section. If you want to see this, press Ctrl-L to see the list file, and then scroll down to the label RESET __PROGSTART; at the end of this section you'll see the OPTION register setting. If you change the FREQ value or interrupt rate you can check back to see that the OPTION register value has in fact changed.

Okay, now let's write the code that runs in the interrupt. Before we do that, though, I want to remind you that in SX/B we must define subroutines and functions before they're called. This creates a bit of a problem if we want to call a subroutine from the interrupt section as the interrupt entry must be the first thing in the program. The solution is actually quite simple: we move the actual code to another location that comes after the subroutine and function declarations. Getting to it is as simple as GOTO. So, the interrupt section of the program ultimately looks like this:

```
  INTERRUPT 1000
    GOTO INT_HANDLER
```

Now it's just a matter of putting the code that runs in the interrupt at a label called INT_HANDLER. In case you're wondering, this section does need to use RETURNINT instead of RETURN, this is necessary to make sure the RTCC value is reloaded properly and interrupts re-enabled. Let's have a look at the interrupt handler:

```
INT_HANDLER:
  IF ops <> M_RUN THEN Next_Digit

Update_Clock:
  INC ms
    IF ms = 10 THEN
    ms = 0
    INC hs
    IF hs = 10 THEN
      hs = 0
      INC ts
      IF ts = 10 THEN
        ts = 0
        INC sec01
        IF sec01 = 10 THEN
          sec01 = 0
          INC sec10
          IF sec10 = 6 THEN
            ops = M_STOP
          ENDIF
        ENDIF
      ENDIF
    ENDIF
  ENDIF
```

```
Next_Digit:
  INC digPntr
  IF digPntr = NumDigits THEN
    digPntr = 0
  ENDIF

Update_Segs:
  Segments = %00000000
  READ Dig_Map + digPntr, DigCtrl
  Segments = display(digPntr)

Check_Finish:
  IF AtFinish = Yes THEN
    ops = M_STOP
  ENDIF

  RETURNINT
```

Yes, it looks a little long, but as you'll soon see, this section of code does most of the work for the race timer. In practice the timer has three modes: 0) stopped and clear, 1) running, and 2) stopped. The current mode is held in the variable called ops (mode and status are SX keywords, so they can't be used). If the timer is not supposed to be running then we skip past its update and move to the next digit of the multiplexed display.

The display update routine points to the next digit (right to left) and then checks to if we need to wrap back to digit zero. Then the segments (anodes) are cleared before reading the current digit pattern from the display array. Clearing the segments before writing a new value to them creates a crisper display to my eye, but you may want to experiment to this. The cathode control value for the current digit is read from a DATA table. While we could have generated the proper active-low cathode control value with code, using a table approach just seemed more elegant.

With the display updated the last thing the interrupt section does is check to see if the finish-line opto-transistor is blocked. If it is, the mode will be set to M_STOP and if the clock was running it will halt at that point, allowing us to view the duration of the race until the Clear button is pressed.

Let's back up – we haven't talked about updating the clock when it's supposed to be running. I used to work for a guy who told me that there are no compromises in product development, but there are choices to be made. Case in point: we could store the timer milliseconds as a word and the timer seconds as a byte, but then we'd have to use division to extract the individual digit values for each position and, as you know, division can be computationally heavy. So, I chose to use discrete variables for each clock digit; this means using five bytes

for the timer instead of three, but I think the benefits far outweigh the use of two additional byte variables. By using this approach we're able to update the display segments much more easily (we'll see that in just a bit) and if we chose to modify the program to send the digits out serially to a terminal, we'd already have the individual digit values in place – again, no division required.

Updating the clock in the interrupt handler is easy; we start with the milliseconds digit, ms. It gets incremented and when it reaches 10 we reset it to zero and increment the hundredths digit, hs. You can see that this process ripples through each of the five variables, the difference being that we don't clear the tens digit when it reach its limit, we simply stop the clock at one minute (60 seconds). The choice of using individual variables to the timer digits does make the code a little longer in this section, but if you look at the assembly output you'll see that there is nearly a 1-for-1 ratio of SX/B-to-assembly so the clock update process is happening pretty quickly.

Now that we have a timer that can updated and display its value, we need to build the control code for starting, stopping, and clearing it, and we'll also need a routine to convert the timer digit values to segment patterns for the LED display. Let's get the program started:

```
Start:
  TRIS_B = %00000111
  PLP_A = %00000011

  COMPARE 1, __PARAM1
```

There's just a couple things going on here – we set the cathode control pins to outputs and pull-up the unused pins on RA. Next, we start the comparator in mode 1. This mode activates the comparator with the result bit output on RB.0. An interesting note here is that we do not need to make RB.0 an output for this pin to operate the LED connected to it; the comparator output bit is connected directly to the pin. The program will monitor the state of RB.0 to determine if the opto-transistor is blocked; when it is the clock will be stopped.

Note, too, that we don't care about the initial output of the COMPARE instruction so we can use one of the internal variables to receive the result. Since the comparator will continue to run and put its result on RB.0 until disabled, we only need to run this instruction one time.

Finally, some may be wondering why we didn't set the TRIS_C register for the segment pins (RC). Well, the PIN definition takes care of that for us when we use the optional OUTPUT directive like this:

```
Segments        PIN     RC      OUTPUT
```

We couldn't do this on RB because we have a mixed I/O structure.

And now we get to the main program loop – which really doesn't have a lot to do.

```
Main:
  DO
    UPDATE_DISPLAY
    IF Go = Yes THEN
      IF ops = M_CLEAR THEN
        ops = M_RUN
      ENDIF
    ENDIF
    IF Clear = Yes THEN
      IF ops = M_STOP THEN
        PUT @ms, 0, 0, 0, 0, 0
        ops = M_CLEAR
      ENDIF
    ENDIF
  LOOP
```

The first thing that happens a is call UPDATE_DISPLAY to convert the timer digit values to segment patterns for the LEDs. Even though we only call this once, I still think it's a good idea to encapsulate into a subroutine so that the program can be somewhat modular. Let's have a look at UPDATE_DISPLAY.

```
UPDATE_DISPLAY:
  READ Seg_Map + ms, display(0)
  READ Seg_Map + hs, display(1)
  READ Seg_Map + ts, display(2)
  READ DP_Map + sec01, display(3)
  IF sec10 = 0 THEN
    display(4) = Blank
  ELSE
    READ Seg_Map + sec10, display(4)
  ENDIF
  RETURN
```

As you can see, this is actually quite simple. READ is used to transfer segment maps from a DATA table into each element of the display array. Since we know where the decimal point is going to be simply hard code that into the program, in this case it will follow the ones digit, and we'll use a separate table with digit patterns plus decimal point – this saves us the step of adding the decimal point bit later. If you decide to modify the timer to have a variable-position decimal point, you could always do something like this:

```
UPDATE_DISPLAY:
```

```
IF DPDigit = 0 THEN
  READ DP_Map + ms, display(0)
ELSE
  READ Seg_Map + ms, display(0)
ENDIF
. . .
```

The one slightly-fancy thing we'll here do is blank the leading zero in the tens digit position, it just makes the output more professional looking in my opinion. From a code standpoint it's a simple matter of clearing the segments when the tens digit is zero, or reading the new segment pattern when it isn't.

To get the timer started it needs to be in mode zero (defined as M_CLEAR). When we get a high input on RA.1 when in this mode the timer is started by updating the ops variable to M_RUN (1). Remember, the interrupt is always running (1000 times each second) so as soon we update ops the display will start changing. Once the car crosses over the finish line and blocks the opto-transistor (which causes the comparator output to go high) the timer will be stopped by changing its mode to M_STOP (2). In this mode we can monitor the Clear input on RA.0 to reset everything.

One of the little-used yet convenient keywords in SX/B is PUT. This command takes a RAM address and a list of one or more values. The first value is written to the address. If there are more values, the address is incremented and subsequent values written. This makes it really easy to move a set of values into a section of contiguous RAM that is not part of an array.

Note that we used the @ (address of) indicator with the ms variable after PUT. We have to do this because PUT is expecting an address as the first parameter. If, however, we use PUT with an array we don't need the @ indicator. The reason for this is that arrays are always treated [internally] as address pointers and offsets.

Putting It Together

Last month I used point-to-point wiring on the Menorah board because most of the hard work was done by Parallax with the Super Carrier. And while this project could be wired point-to-point, I certainly don't have the patience to do it. Enter ExpressPCB. Since I don't create a lot of printed circuit boards I find the ease-of-use and ordering via ExpressPCB.com to be right up my alley. I particularly like that the companion program, ExpressSCH (schematic capture), can be linked to the board file to assist in making connections – this was especially useful for the 7-segment displays.

I'll never be accused of being a PCB layout expert, so I'm not going to spend a great deal of time here. What I want to share with you was my solution for dealing with the displays. I started by selecting display modules that have rows of horizontal pins. Once I created a custom component in ExpressPCB and dropped five of them onto the board, I found the easiest way to get the segment signals to all was to lay down a horizontal buss of eight lines on the top side (red) of the PCB. Each segment is connected to its respective buss lines from the bottom side (green) of the board with a via. A via looks like a pad, but it's smaller and its purpose is to route a signal from one side of the board to the other. After the segments were connected to the buss the segment resistors were connected. This was the only tricky part of the board layout as traces pass between pads – no worries, though, there is plenty of room and unless one is very clumsy with a soldering iron there is little possibility of solder bridges. Figure 141.3 shows the timer layout using the standard Mini-Board form factor.

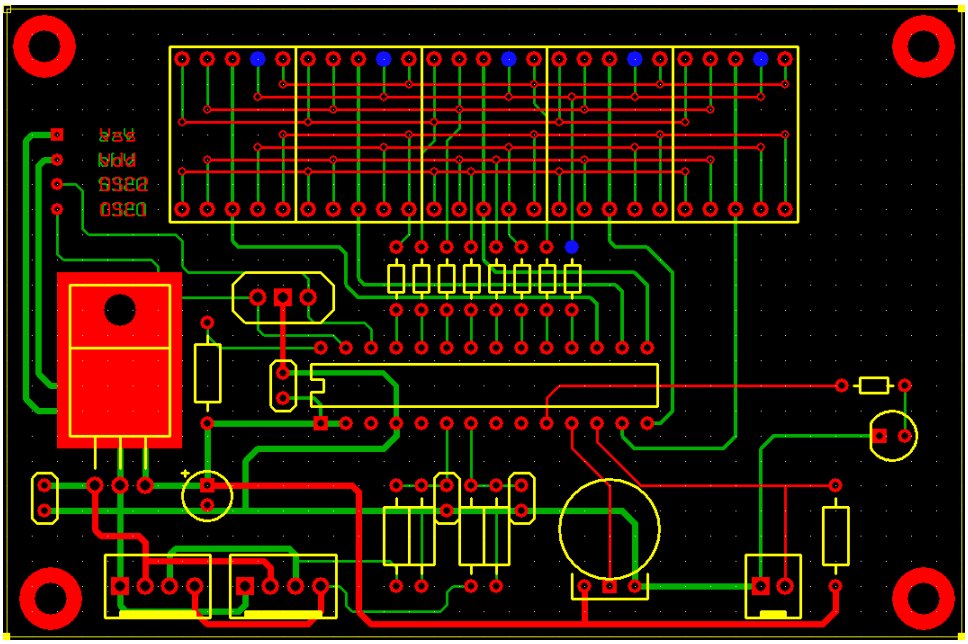

Figure 141.3: Timer PCB Layout

If you haven't used ExpressSCH and ExpressPCB do give them a try – they're free, and if you can fit your project into their Mini-Board format (as we did here) you can get three PCBs for $62 and have them back three business days after you place the order. One thing I want to encourage you to do is learn to use ExpressSCH to create your schematics before moving on

to ExpressPCB to layout the board. I know, we're all in a time crunch but believe me, putting your project into ExpressSCH first will save you a lot of headaches. First, it will check all your connections and warn you of possible problems. Second, you can connect ExpressPCB to the ExpressSCH so that making connections on the board is much easier. You can see in Figure 141.3 that the pads in blue are supposed to be connected – I promise that this feature will save you lots of trouble and you'll be happy you spent the extra time with ExpressSCH.

Construction was straightforward. Like most, I start with the low-profile components first (resistors) and work my way up to the taller components. The connectors are soldered on the back, and I didn't actually put a connector into J3 (SX-Key/SX-Blitz); I simply used pads with small holes to make holding the Key/Blitz (equipped with a male-male header) against the board a little easier. Figures 141.4 and 141.5 show the front and back of my prototype PCB.

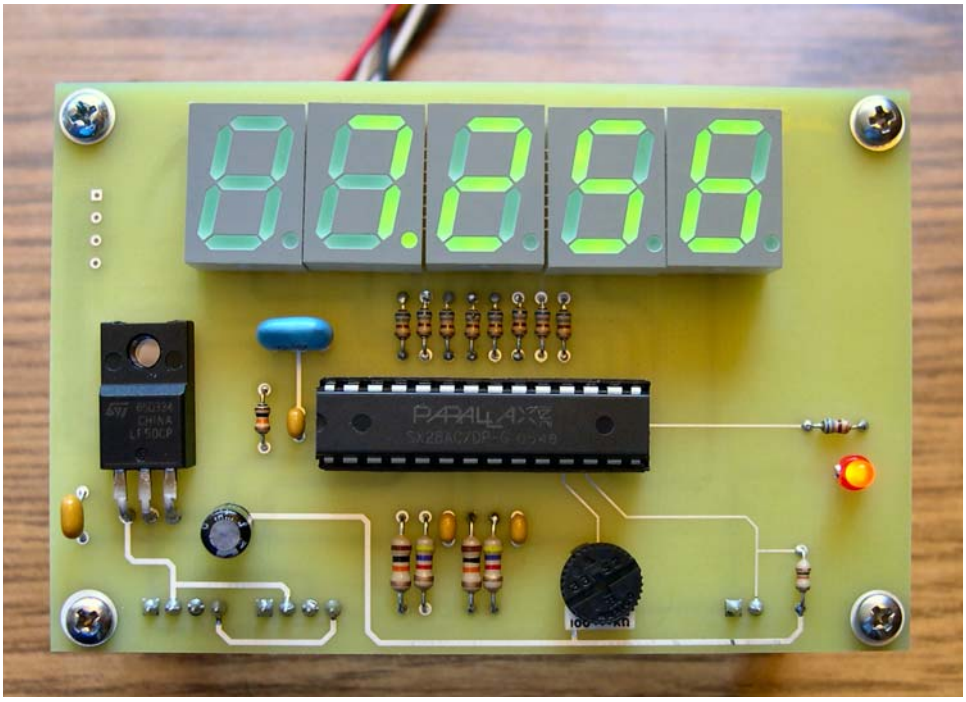

Figure 141.4: Timer PCB Front

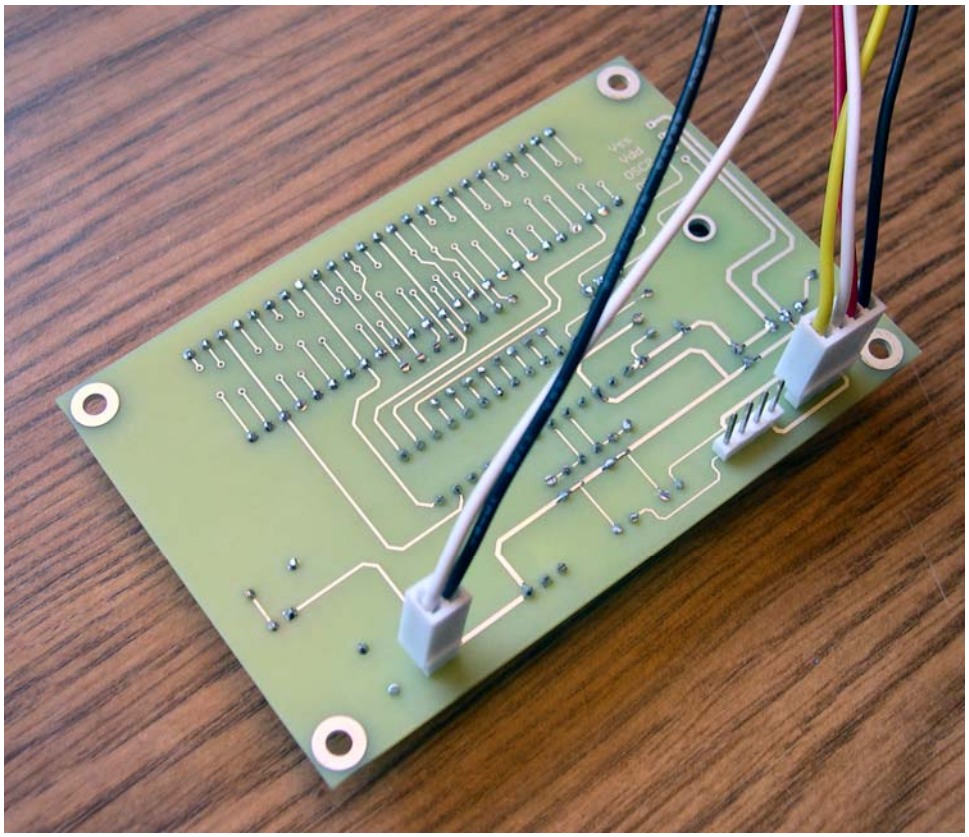

Figure 141.5: Timer PCB Back

Let's Race!

To put this project to work in a Pinewood Derby race track you'll need to mount an LED source in the track at the finish line and the photo-transistor receiver above it (I used a source/detector pair from RadioShack). In my experiments with the timer I found it works best when the photo-transistor is shielded with a plastic tube. With everything in place and powered, adjust R16 until the finish LED comes on, then back off until it goes off. Make sure that the LED goes full off – if it looks a bit dim then the comparator threshold has been very

near the opto/10K junction voltage and the output is oscillating between on and off. Back R16 off a bit more until the LED is full off.

Figure 141.6 shows a suggested master controller for single- or multi-track setups. There's not much too this so it could be wired point-to-point. Don't leave out the diodes if you're going to control a start solenoid from the power source used for the timer(s). The diodes will protect the power supply and control inputs to the timer(s) from any inductive kickback produced by the solenoid.

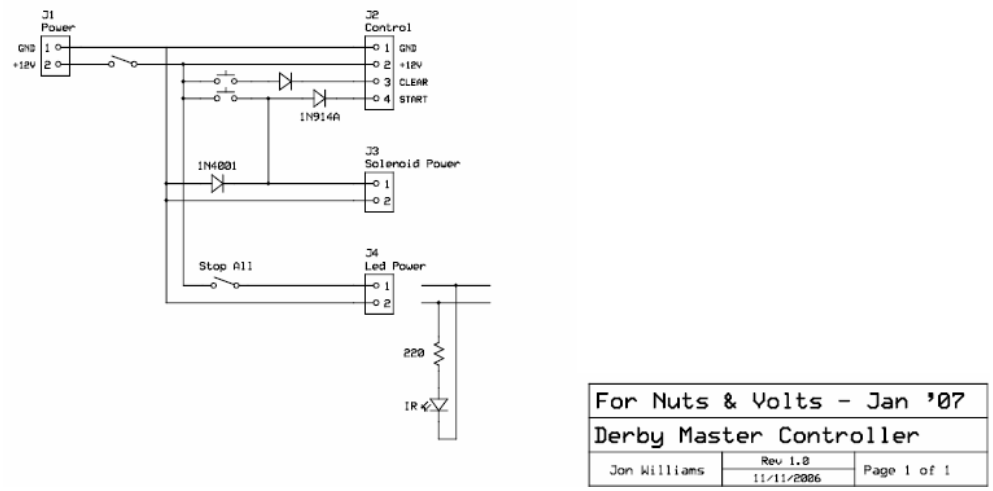

Figure 141.6: Timer PCB Back

Okay, it's time to get your Pinewood racers out and get racing. With one millisecond resolution, you hardcore racers will have not trouble determining which car changes work and which don't.

Until next time, Happy Stamping.

Resources

Jon Williams
jwilliams@efx-tek.com

Parallax, Inc.
www.parallax.com

ExpressSCH/ExpressPCB
www.expresspcb.com

Pinewood Derby Timer Bill of Materials		
Designator	**Value**	**Source**
C1	0.47	Mouser 80-C320C474M5U
C2	47	Mouser 647-UVR1C470MDD
C3-C5	0.1	Mouser 80-C315C104M5U5TA
D0-D4	7-segment, CC	Mouser 859-LTS-5603AG
D6		Mouser 638-204HT
J1, J2		Mouser 571-6404544
J4		Mouser 571-6404542
Q1		RS 276-142
R1, R3, R5, R14		Mouser 291-10K-RC
R2, R4	4.7K	Mouser 291-4.7K-RC
R6-R14	470	Mouser 299-470-RC
R16	100K	Mouser 652-3352T-1-104LF
U1	5v, LDO	Mouser 511-LF50CP
U2	SX28AC/DP	Parallax SX28AC/DP-G
X4		Mouser 538-22-01-2027
XR1	20 MHz	Parallax 250-02060

Source Code

```
' =============================================================================
'
'   File..... TRACK_TIMER.SXB
'   Purpose... Pinewood Derby Track Timer
'   Author.... Jon Williams, EFX-TEK
'   E-mail.... jwilliams@efx-tek.com
'   Started...
'   Updated... 17 NOV 2006
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' Track timer for pinewood derby racing.  The Clear and Go inputs can
' be connected to other devices so that all are under control from a
' single master race controller in a multi-track setup.


' -----------------------------------------------------------------------------
' Conditional Compilation Symbols
' -----------------------------------------------------------------------------


' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE          SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ            20_000_000
ID              "Derby1.0"


' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

Clear           PIN     RA.0
Go              PIN     RA.1

AtFinish        PIN     RB.0                    ' comparitor output bit

DigCtrl         PIN     RB
Segments        PIN     RC      OUTPUT


' -----------------------------------------------------------------------------
' Constants
```

```
' -----------------------------------------------------------------------------

Yes             CON     1                       ' for active-high inputs
No              CON     0

M_CLEAR         CON     0                       ' clock clear and stopped
M_RUN           CON     1                       ' clock running
M_STOP          CON     2                       ' clock stopped

Blank           CON     %00000000               ' all segments off
DPoint          CON     %10000000               ' DP in bit7
DPDigit         CON     3                       ' DP after secs digit

NumDigits       CON     5                       ' for 5-digit display


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

ops             VAR     Byte                    ' operational mode

ms              VAR     Byte                    ' milliseconds digit
hs              VAR     Byte                    ' hundredths digit
ts              VAR     Byte                    ' tenths digit
sec01           VAR     Byte                    ' ones digit
sec10           VAR     Byte                    ' tens digit

digPntr         VAR     Byte                    ' digit pointer
display         VAR     Byte(NumDigits)         ' current display segments


' -----------------------------------------------------------------------------
  INTERRUPT 1000                                ' run every millisecond
' -----------------------------------------------------------------------------

  GOTO INT_HANDLER


' =============================================================================
  PROGRAM Start
' =============================================================================


' -----------------------------------------------------------------------------
' Subroutine Declarations
' -----------------------------------------------------------------------------

UPDATE_DISPLAY SUB      0                       ' convert digits to segments
```

```
' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Start:
  TRIS_B = %00000111                          ' make dig ctrl pins outs
  PLP_A = %00000011                           ' pull up unused pins

  COMPARE 1, __PARAM1                         ' activate comparitor

Main:
  DO
    UPDATE_DISPLAY                            ' refresh segment array
    IF Go = Yes THEN                          ' go button pressed
      IF ops = M_CLEAR THEN                   ' were we cleared?
        ops = M_RUN                           ' yep, so we can run
      ENDIF
    ENDIF
    IF Clear = Yes THEN                       ' clear button pressed?
      IF ops = M_STOP THEN
        PUT @ms, 0, 0, 0, 0, 0
        ops = M_CLEAR                         ' clear the clock
      ENDIF
    ENDIF
  LOOP


' -----------------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------------

INT_HANDLER:
  IF ops <> M_RUN THEN Next_Digit             ' skip update if stopped

Update_Clock:
  INC ms                                      ' inc milliseconds digit
  IF ms = 10 THEN
    ms = 0
    INC hs                                    ' inc hundredths digit
    IF hs = 10 THEN
      hs = 0
      INC ts                                  ' inc tenths digit
      IF ts = 10 THEN
        ts = 0
        INC sec01                             ' inc seconds digit
        IF sec01 = 10 THEN
          sec01 = 0
          INC sec10                           ' inc tens digit
          IF sec10 = 6 THEN                   ' one minute maximum time
            ops = M_STOP                      ' stop the clock
          ENDIF
```

```
        ENDIF
      ENDIF
    ENDIF
  ENDIF

Next_Digit:
  INC digPntr                                  ' point to next digit
  IF digPntr = NumDigits THEN                  ' check pointer
    digPntr = 0                                ' wrap if needed
  ENDIF

Update_Segs:
  Segments = %00000000                         ' blank segments
  READ Dig_Map + digPntr, DigCtrl              ' update digit control
  Segments = display(digPntr)                  ' output new segments

Check_Finish:
  IF AtFinish = Yes THEN                       ' if car at finish line
    ops = M_STOP                               ' stop clock
  ENDIF

  RETURNINT

' -----------------------------------------------------------------------------

UPDATE_DISPLAY:
  READ Seg_Map + ms,    display(0)             ' update segment maps
  READ Seg_Map + hs,    display(1)
  READ Seg_Map + ts,    display(2)
  READ DP_Map  + sec01, display(3)

  IF sec10 = 0 THEN
    display(4) = %00000000                     ' blank leading zero
  ELSE
    READ Seg_Map + sec10, display(4)
  ENDIF

  RETURN


' =============================================================================
' User Data
' =============================================================================

Seg_Map:                                       ' segments maps
'        .gfedcba
  DATA  %00111111                                ' 0
  DATA  %00000110                                ' 1
  DATA  %01011011                                ' 2
  DATA  %01001111                                ' 3
  DATA  %01100110                                ' 4
```

```
  DATA  %01101101                                ' 5
  DATA  %01111101                                ' 6
  DATA  %00000111                                ' 7
  DATA  %01111111                                ' 8
  DATA  %01100111                                ' 9

DP_Map:                                         ' segments maps with DP
'        .gfedcba
  DATA  %10111111                                ' 0.
  DATA  %10000110                                ' 1.
  DATA  %11011011                                ' 2.
  DATA  %11001111                                ' 3.
  DATA  %11100110                                ' 4.
  DATA  %11101101                                ' 5.
  DATA  %11111101                                ' 6.
  DATA  %10000111                                ' 7.
  DATA  %11111111                                ' 8.
  DATA  %11100111                                ' 9.


Dig_Map:                                        ' digit select map
  DATA  %11110000
  DATA  %11101000
  DATA  %11011000
  DATA  %10111000
  DATA  %01111000
```