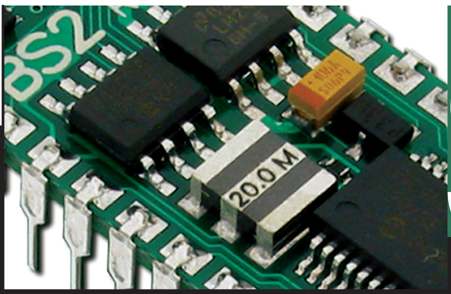


■ BY JON WILLIAMS

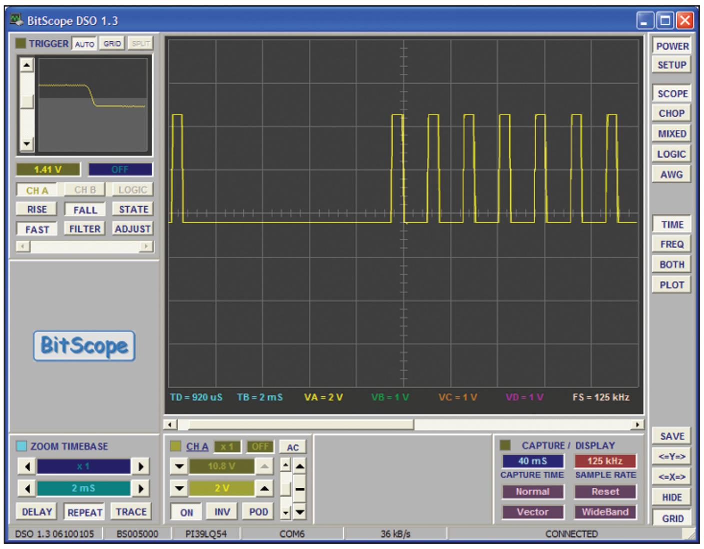
MORE SURPLUS SUCCESS

I MAY HAVE MENTIONED MY FRIEND BRIAN once or twice. Brian's a great guy — a super smart IT professional by day and a bring-down-the house DJ by night. When I lived in Dallas, Brian was a tad jealous because I had Tanner (geek heaven) within minutes of my home. So, I move back to Los Angeles a couple years ago and no more Tanner for me (and I miss them). But, what do I have? That's right! All Electronics — another gate into geek heaven. Brian was beside himself; what luck I have with my proximity to fantastic suppliers. Both Tanner and All are great about stocking new products as well as a boatload of interesting surplus. One of the cooler products that All carries — and at a ridiculously low price — is the VEX Robotics Transmitter and Receiver Add-on kit; brand new and in the box. You just can't beat that.

Until you get home, that is, and find out that the receiver in the box is not capable of [directly] driving servos. What? You see, the little receiver box was, in fact, designed to be interfaced with the VEX controller, so it simply outputs a continuous stream of servo pulse width data. Figure 1 shows what the output looks like on a 'scope.

The pulses are active-low, and each is preceded by a framing pulse that is about 500 microseconds wide. Notice that one of the pulses is very wide relative to the others; nearly nine milliseconds. This is the sync pulse and by finding this, we can get the position data to the correct servo. It turns out that using the PPM (pulse position modulation) is pretty easy. After locating the sync pulse, we simply wait for a high-going edge and turn on the first servo. We leave this output on until the signal has dropped and goes back high again; this is the signal to move to the next servo. This process continues for six channels.

■ FIGURE 1. VEX PPM Stream.



The VEX transmitter has two joysticks that give complete analog control over channels one through four, and two push buttons each for channels five and six. With the last two channels, the servo pulse width will be 1.5 milliseconds (center) with neither button pressed. If the top button is pressed, the servo output drops to 1.0 milliseconds; if the bottom button is pressed, the servo output bumps up to 2.0 milliseconds. So while we can control servos with channels five and six, these channels are limited to three servo positions.

The circuit for converting the PPM stream to usable servo outputs couldn't be much simpler — and you can see this in Figure 2. This is a very generic SX circuit with a power supply, and connection for the receiver, and header for the servos. There are two jumpers on the board: one for selecting the servo power (+5V DC or Vin), and one for setting the behavior of servo outputs five and six.

DECODING THE PPM STREAM

The first thing we need to do with the PPM stream is locate the sync pulse. I measured this to be about 8.9

milliseconds in duration. That said, all we have to do is wait for a low-going pulse that is longer than a servo position pulse; this will let us know we've found the sync pulse.

```
SUB WAIT_SYNC
  pulseTmr = 0
  DO WHILE PPM = 0
    PAUSEUS 10
    INC pulseTmr
  LOOP
  IF pulseTmr < 400 THEN WAIT_SYNC
  ENDSUB
```

The subroutine called WAIT_SYNC takes care of this. The routine starts by clearing a timer variable (*pulseTmr*) and then dropping into a loop that monitors the PPM input for being low. As long as this input stays low, the timer will be incremented every 10 microseconds. Note that timing doesn't have to be super precise here; all we're looking for is a low pulse that couldn't be a position value.

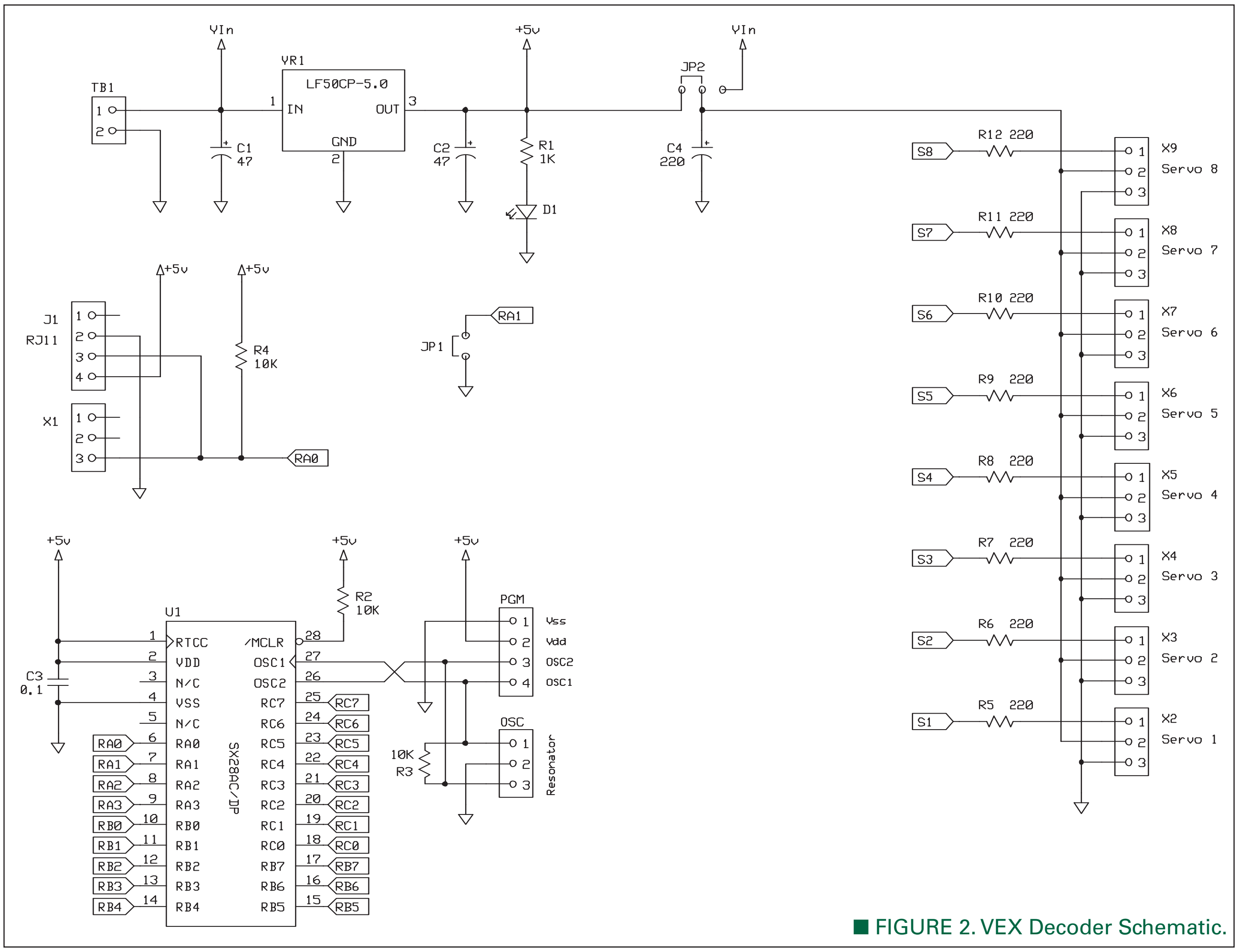
When the PPM line goes high, loop terminates and the timer value is checked; if we find a pulse greater than about four milliseconds, we know that we have sync and we can return to the caller. If we happen to catch a position pulse, the routine will run again.

On start-up, we'll clear the servo outputs and then check the mode input jumper. As RA.1 has the internal pull-up enabled, we'll see a "1" on RA.1 when in standard servo mode, or a "0" when in what I'm calling "servo plus" mode. Let's look at standard mode first.

```
Start:
  SvoPort = %00000000

Main:
  IF MJumper = 0 THEN Servo_Plus

' ----------
' Standard servo control
' ----------
'
Standard_Servos:
  WAIT_SYNC
  svoPntr = %0000_0001
  DO
    SvoPort = svoPntr
    WAIT_HI_LO
    WAIT_LO_HI
    svoPntr = svoPntr << 1
  LOOP UNTIL svoPntr = %0100_0000
  GOTO Start
```



■ FIGURE 2. VEX Decoder Schematic.

After waiting for the sync pulse, an internal servo pin pointer (*svoPntr*) is set to %00000001 to activate the first servo when applied to port RC. We drop into a loop where the pointer is written to the port, the program waits for the 500 microsecond framing pulse to end (high), and then waits on the timing pulse (low) to finish. To keep the listing neat, I wrote a couple dirt-simple subroutines for waiting on the edge transitions:

```
SUB WAIT_LO_HI
  DO WHILE PPM = 0
  LOOP
  ENDSUB

SUB WAIT_HI_LO
  DO WHILE PPM = 1
  LOOP
  ENDSUB
```

I deliberately made this program *no assembly required*, but if you're comfortable with assembly you could easily substitute an embedded instruction. Remember that SX/B allows the insertion of a single line of assembly code by prefacing the line with the backslash character. So, instead of WAIT_LO_HI, we could use:

```
  \ JNB PPM, @$
```

And instead of WAIT_HI_LO we could use:

```
  \ JB PPM, @$
```

The @$ means to jump to the current address until the bit changes.

Back to the servo loop. After the framing and timing pulses are finished, the servo pointer is shifted left for the next servo. Once we have shifted this bit to Bit 6 of the variable, the loop terminates and the program jumps back to the top. Yes, the board has eight servo outputs (you'll see why in a bit) but their VEX transmitter only provides data for six. If you happen to find a device with a similar PPM output that handles eight channels, the code is easily modified.

SERVOS PLUS

I mentioned earlier that the VEX transmitter has two push buttons [each] for channels five and six — in standard servo mode, this limits the servo positions for channels five and six to the center and to either extreme (left and right). What if we had a robot or animatronic that required four or less servos and we wanted to use channels five and six as digital control outputs? How could we do this?

Handling the first four servos is similar to what we've just done. For channels five and six, we're going to measure the timing pulse. If that pulse is about 500 microseconds, it means the top button for the channel was pressed and we can turn the corresponding output on. If the pulse is about 1,500 microseconds, that means the bottom button was pressed and we'll turn the corresponding output pin off. The only other possibility is that we measure about 1,000 microseconds; in this case, we will do nothing with the output.

```
' ----------
' Four servos + two on/off
' ----------
'
Servo_Plus:
  WAIT_SYNC
  svoPntr = %0000_0001
  DO
    SvoPort = SvoPort | svoPntr
    WAIT_HI_LO
    WAIT_LO_HI
    svoPntr = svoPntr << 1
    SvoPort = SvoPort & %0011_0000
  LOOP UNTIL svoPntr = %0001_0000

Ctrl_Port1:
  WAIT_HI_LO
  pulseTmr = 0
  DO WHILE PPM = 0
    PAUSEUS 10
    INC pulseTmr
  LOOP

  IF pulseTmr < 60 THEN
    Control1 = IsOn
  ELSEIF pulseTmr > 110 THEN
    Control1 = IsOff
  ENDIF

Ctrl_Port2:
  WAIT_HI_LO
  pulseTmr = 0
  DO WHILE PPM = 0
    PAUSEUS 10
    INC pulseTmr
  LOOP

  IF pulseTmr < 60 THEN
    Control2 = IsOn
  ELSEIF pulseTmr > 110 THEN
    Control2 = IsOff
  ENDIF

  GOTO Main
```

The servo portion of the loop starts out as before, but note now that instead of simply writing the value of *svoPntr* to the output port, we are ORing with the port. The reason we have to do this is to protect what is presently sitting on the output bits corresponding to channels five and six. Note, too, that there's one more line after the pointer is updated. This line clears the servo output that just ran while maintaining whatever happens to be sitting on channels five and six.

After completing the servos, the low-going timing pulses of channels five and six are measured and the output is updated as determined by the pulse. Pretty simple, really, and pretty darned useful.

So there we have it: a simple SX circuit that will turn that $30 VEX add-on kit into something that can actually drive servos and digital outputs. One last note before we move on. As the timing is controlled by the VEX transmitter, we can actually run this circuit using the internal 4 MHz clock source. If you do this, you can leave R3, the OSC socket, and the resonator off the board. I put them onto mine so I have options — you can see in the photo of the completed board (Figure 3) that R3 and the socket are installed, but the resonator is not.

DOUBLE IT UP

Having such a svelte circuit leaves us with a bit of a dilemma when using ExpressPCB's mini-board service: There's a ton of unused board space. Should we let this go to waste? Absolutely not! Let's double it up. When I started laying out the board, I found that the circuit would comfortably fit in half the space of a standard mini-board. Excellent — let's just copy-and-paste and get two boards for the price of one.

Not so fast, there, chief. Before we double-up any of your boards, we need to do a thorough check of the layout using a link to the schematic. This will save us a lot of trouble later; not all (as I found out), but most. Save the single board file separately so you can come back and update it if necessary.

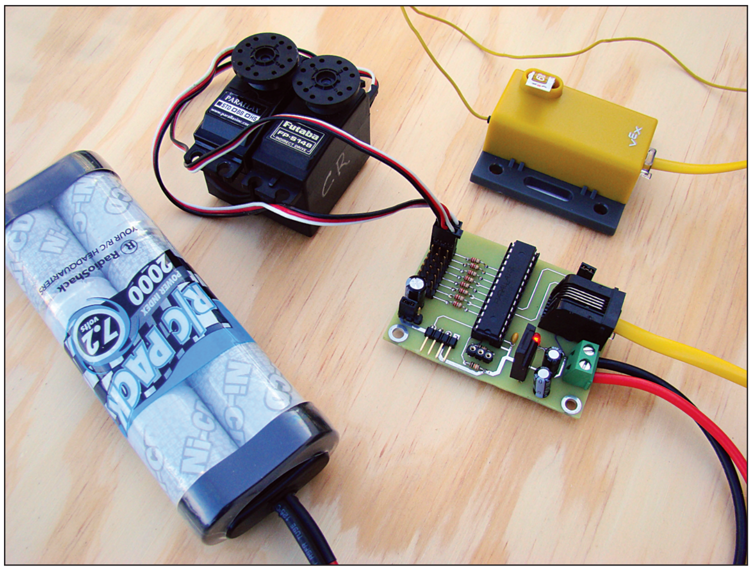
I did, and here's why. While having lunch with my "networking" pal, Peter, he talked about making generic boards as generic as possible, and this really is the case with this board. It dawned on me — especially having just written a servo animation driver for the Prop-SX — that I could add another connector and make this board a standard servo controller.

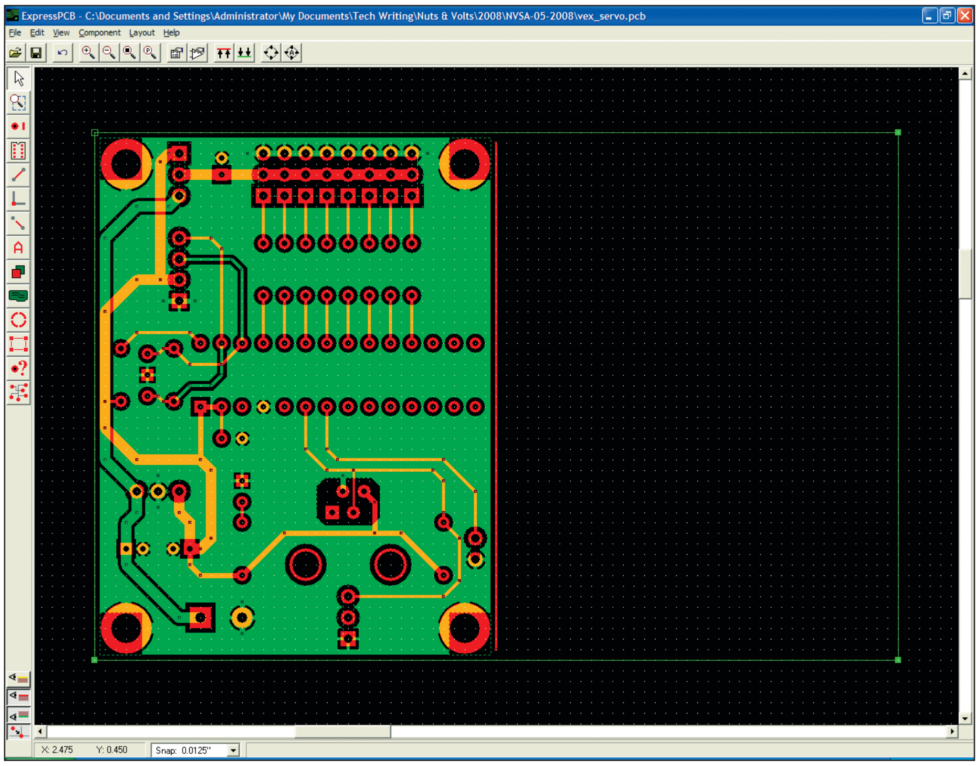
If you look closely at the layout, you'll see that the RJ-11 sits on top of a three-pin header; this allows me to stuff the board two different ways based on what I want. The RJ-11 allows me to make the standard VEX decoder, or use phone cable for my input. If I want to create a standard servo controller for a BASIC Stamp or SX project, I'll replace the RJ-11 with a three-pin servo header.

Figure 4 shows a screenshot from ExpressPCB with the completed layout for one board. After this file is saved, it's a simple matter of copy, paste, and then adjust position (while everything is still highlighted) of the duplicate parts. Save the double board as a separate file. And note that once we've doubled things, using the "Highlight Net Connections" tool is no longer functional as we have duplicated part numbers.

Since we did a "background" servo driver last May I won't go into that, but what I will show you is how I created the servo animation driver I mentioned earlier. Many artists use a program called VSA (Visual Show Automation) for running props and servo-based animatronic displays. VSA allows one to integrate servo movement and sound very easily, and has become a favorite, especially with its low price (about $50).

VSA uses the SEETRON (Scott Edwards) MiniSSC protocol as its default. Being a very clever guy, Scott made


■ FIGURE 3. VEX Decoder Ready For Testing.


ExpressPCB - C:\Documents and Settings\Administrator\My Documents\Tech Writing\Nuts & Volts\2008\NVSA-05-2008\vex_servo.pcb

X: 2.475 Y: 0.450 Snap: 0.0125"

■ FIGURE 4. Single Board Layout.

◗ PARTS LIST

Item	Description	Supplier/Part No.
◗ C1-C2	47 µF	Mouser/647-UVR1V470MDD
◗ C3	0.1 µF	Mouser/80-C315C104M5U
◗ C4	220 µF	Mouser/647-UVR1C221MED
◗ D1	3 mm LED	Mouser/859-LTL-4222N
◗ J1	RJ11	Mouser/571-520250-2
◗ JP1-JP2	0.1” pin strip header	Mouser/517-6111TG
◗ Jumpers	0.1” shunt	Mouser/151-8000-E
◗ OSC*	0.1” pin socket	Mouser/506-510-AG90D
◗ PCB		ExpressPCB.com
◗ PGM	0.1” R/A header	Mouser/517-5111TG
◗ R1	1K	Mouser/299-1K-RC
◗ R2, R4	10K	Mouser/299-10K-RC
◗ R3*	10K	Mouser/299-10K-RC
◗ R5-R12	220 Ω	Mouser/299-220-RC
◗ Resonator*	4 MHz	Parallax/250-04050
◗ Resonator*	20 MHz	Parallax/250-02060
◗ Resonator*	50 MHz	Parallax/250-05060
◗ Socket	28 pin	Mouser/571-1-390-261-9
◗ TB1	Terminal block, 5 mm	Mouser/571-2828362
◗ U1	SX28	Parallax/SX28AC/DP
◗ VR1	LF50CP-5.0	Mouser/511-LF50CP
◗ X1-X9	0.1” pin strip header	Mouser/517-6111TG

* = Optional components

Note: JP1, JP2, and X1-X9 are cut from a single 40-pin part.

the protocol simple; to change the position of a servo, the host will send three bytes to the controller: sync, servo number, position. By using a virtual UART and servo driver, the foreground program for a MiniSSC-compatible servo controller becomes downright trivial:

```
Start:
  ' center servos
  PUT pos, 150, 150, 150, 150, 150, 150, 150,
150

Main:
  sync = RX_BYTE
  IF sync <> 0xFF THEN Main

  chan = RX_BYTE
  value = RX_BYTE

Process_Value:
  IF chan < 8 THEN
    value = value MIN LO_LIMIT
    value = value MAX HI_LIMIT
    pos(chan) = value
  ENDIF

  GOTO Main
```

Yes, that’s it. At Main, we monitor the input stream until a 0xFF shows up; the next two bytes are the servo number and position, respectively. If the servo number is valid, the position gets checked against hard position limits (to prevent servo damage) and written to the servo driver. My friends in the Dallas Personal Robotics Group have a saying: *It's harder than it looks*. In this case, however, it really isn't.

There's a great lesson here: We shouldn't be afraid to explore trails blazed by others to see if we might learn what they did. For example, why did Scott select 0xFF as the sync value? Because — with the position units used — that would never be a valid position value. I know that this seems terribly obvious, and yet I want to encourage you not to take the simple things for granted. Many of us do and that leads to unnecessary complications. Whenever possible, keep things simple. Simple is fun. Simple is elegant. Simple is [usually] robust.

Okay, it's your turn now. There is still a bit of space on the board — even the half board — and a useful exercise might be to add IDC-style headers so that you can access all of the RA and RB pins; this would make the board truly generic. It doesn't cost anything but time to experiment with ExpressPCB, so why not give it a try? Even if you don't build the servo board, what you learn will pay off in future projects.

Until next time — Happy Stamping, SX style! **NV**

RESOURCES

JON WILLIAMS
jwilliams@efx-tek.com

PARALLAX, INC.
www.parallax.com

ALL ELECTRONICS
www.allelectronics.com

EXPRESSPCB
www.expresspcb.com

VISUAL SHOW AUTOMATION
www.brookshire software.com