



Column 149, May 2008 by Jon Williams:

More Surplus Successes

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 Tanners (geek Heaven) within minutes of my home. So I move back to Los Angeles back a couple years ago and no more Tanners 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 Tanners and All are great about stocking new product 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 149.1 shows what the output looks like on a 'scope.

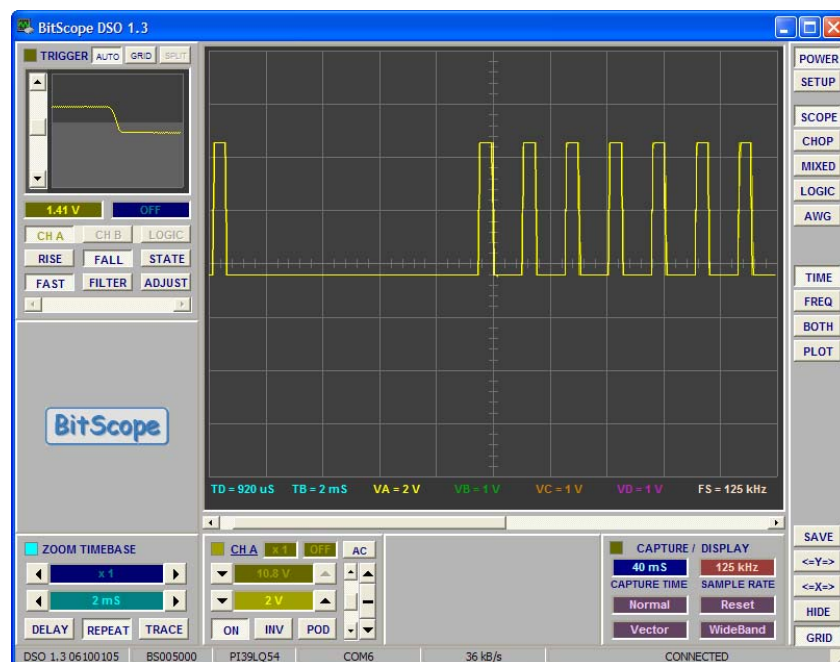


Figure 149.1

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 had dropped and goes back high again; this is the signal to move to the next servo. This process continues for six channels.

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 149.2. 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 (+5vdc or Vin), and one for setting the behavior of servo outputs five and six.

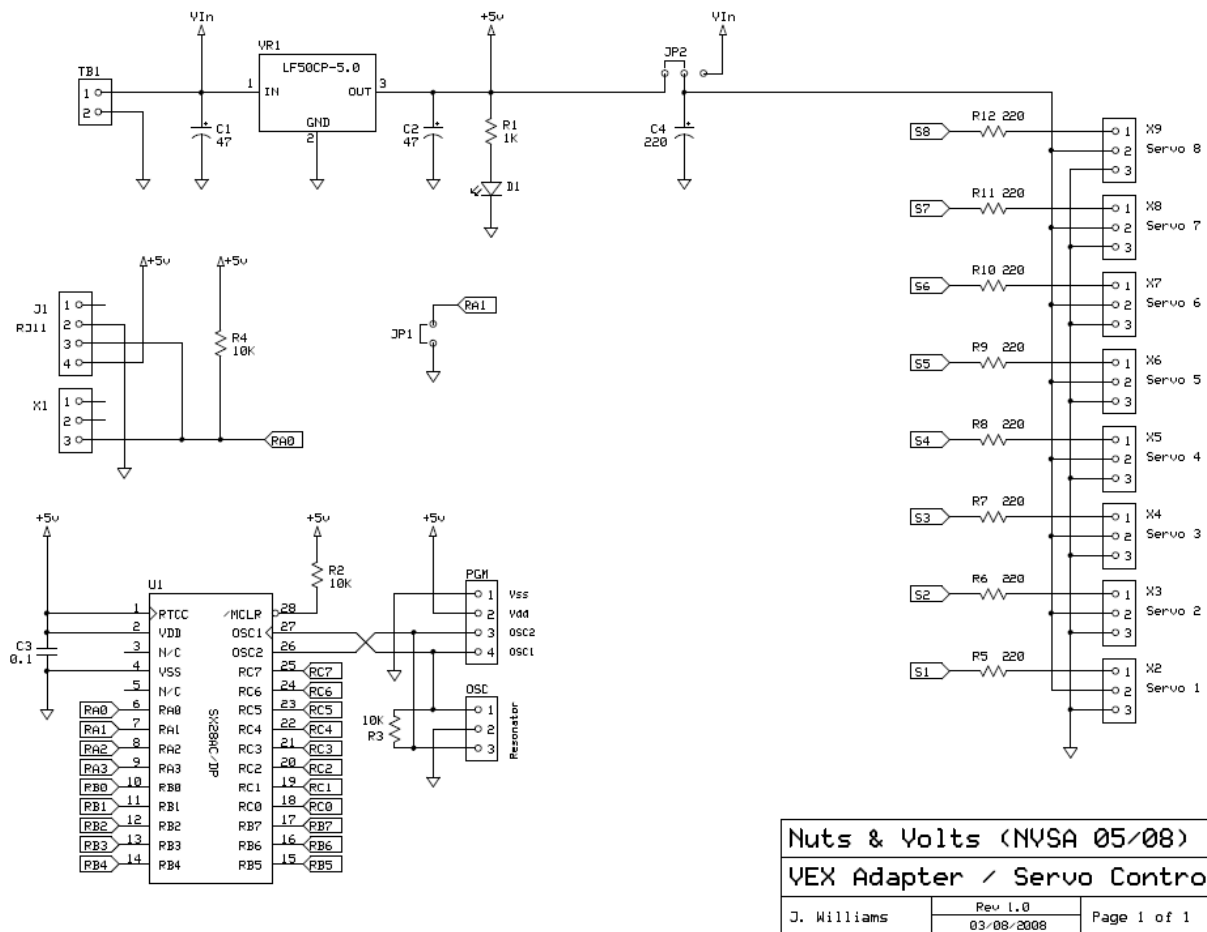


Figure 149.2

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 we 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, as 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 4 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 startup 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
```

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 and 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 shift this bit to Bit6 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 there 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 1500 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 1000 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  
    Controll = IsOn  
  ELSEIF pulseTmr > 110 THEN  
    Controll = 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 picture of the completed board (Figure 3) that R3 and the socket are installed, but the resonator is not.

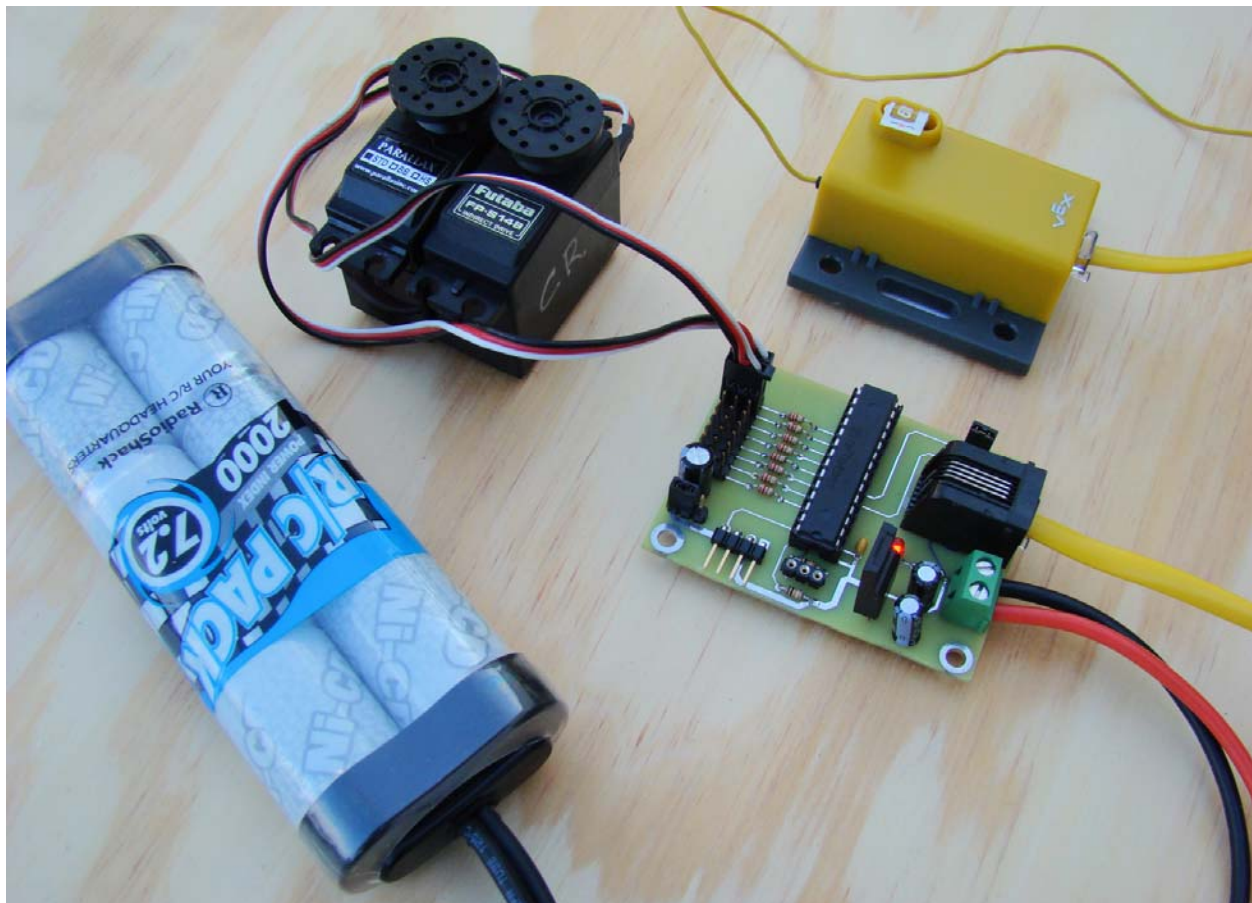


Figure 149.3

Double It Up

Having such a svelte circuit leaves us with a bit of a dilemma when using ExpressPCBs 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, their, 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 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 3-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 3-pin servo header.

Figure 4 shows a screen shot 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 double things, using the "Highlight Net Connections" tool is no longer functional as we have duplicated part numbers.

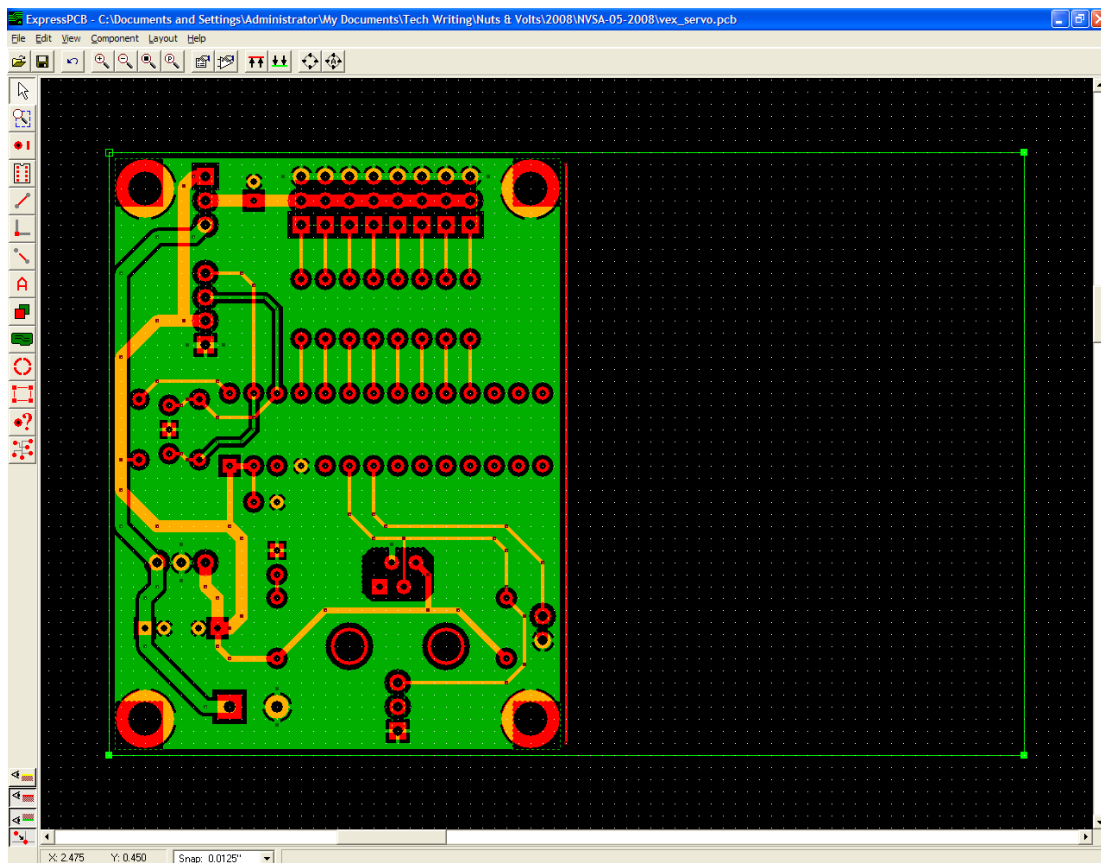


Figure 149.4

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 animatronics 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 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, 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 they 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 access to 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!

Resources

All Electronics
www.allelectronics.com

ExpressPCB
www.expresspcb.com

Visual Show Automation
www.brookshiresoftware.com

Parts List for the VEX Decoder/Servo Driver		
Designator	Value	Source
C1-C2	47	Mouser 647-UVR1V470MDD
C3	0.1	Mouser 80-C315C104M5U
C4	220	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		Mouser 517-5111TG
R1	1 K	Mouser 299-1K-RC
R2, R4		Mouser 299-10K-RC
R3*		Mouser 299-10K-RC
R5-R12		Mouser 299-220-RC
Resonator*		Parallax 250-04050
Resonator*		Parallax 250-02060
Resonator*		Parallax 250-05060
Socket	28 pin	Mouser 571-1-390-261-9
TB1		Mouser 571-2828362
U1		Parallax SX28AC/DP
VR1		Mouser 511-LF50CP
X1-X9		Mouser 517-6111TG

* = Optional components.

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

Code Listing

```

' =====
'
' File..... Servo8.SXB
' Purpose...
' Author.... Jon Williams, EFX-TEK
'           Copyright (c) 2008 EFX-TEK
'           Some Rights Reserved
'           -- see http://creativecommons.org/licenses/by/3.0/
' E-mail.... jwilliams@efx-tek.com
' Started...
' Updated... 01 MAR 2008
'
' =====
'
' -----
' Program Description
' -----
'
' The protocol is identical to the SEETRON (www.seetron.com) MiniSSC:
'
' <sync><channel><value>
'
' sync..... 0xFF (255)
'
' channel... 0 to 7 for servos
'
' value..... 55 to 245 for servos (150 = center)

```

```

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

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

DEVICE          SX28, OSCHS2, TURBO, STACKX, OPTIONX, BOR42
FREQ            50_000_000
ID              "Servo8"

' -----
' I/O Pins
' -----

RX              PIN      RA.0 INPUT          ' true mode serial input

UndefinedRA1    PIN      RA.1 INPUT PULLUP
UndefinedRA2    PIN      RA.2 INPUT PULLUP
UndefinedRA3    PIN      RA.3 INPUT PULLUP

UndefinedRB     PIN      RB   INPUT PULLUP

ServoCtrl       PIN      RC
Servo0          PIN      RC.0 OUTPUT
Servo1          PIN      RC.1 OUTPUT
Servo2          PIN      RC.2 OUTPUT
Servo3          PIN      RC.3 OUTPUT
Servo4          PIN      RC.4 OUTPUT
Servo5          PIN      RC.5 OUTPUT
Servo6          PIN      RC.6 OUTPUT
Servo7          PIN      RC.7 OUTPUT

' -----
' Constants
' -----

IsOn            CON      1
IsOff           CON      0

Yes             CON      1
No              CON      0

LO_LIMIT        CON      55          ' to prevent servo burn-up
HI_LIMIT        CON      245

' Bit dividers for 3.255 uS interrupt

Baud2400        CON      128
Baud4800        CON      64
Baud9600        CON      32
Baud19K2        CON      16
Baud38K4        CON      8

Baud1x0         CON      Baud38K4          ' 1 bit period (ISR counts)
Baud1x5         CON      Baud1x0 * 3 / 2      ' 1.5 bit periods

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

flags           VAR      Byte          ' (keep global)
  isrFlag       VAR      flags.0
  rxReady       VAR      flags.1          ' rx byte waiting

```

```

sync          VAR      Byte
chan          VAR      Byte
value        VAR      Byte

rxSerial      VAR      Byte (16)
rxBuf         VAR      rxSerial(0)          ' 8-byte buffer
rxCount       VAR      rxSerial(8)          ' rx bit count
rxDivide      VAR      rxSerial(9)          ' bit divisor timer
rxByte        VAR      rxSerial(10)         ' received byte
rxHead        VAR      rxSerial(11)         ' buffer head (write to)
rxTail        VAR      rxSerial(12)         ' buffer tail (read from)
rxBufCnt      VAR      rxSerial(13)         ' # bytes in buffer

svoData       VAR      Byte (16)           ' bank servo data
pos           VAR      svoData(0)          ' position table
pos0          VAR      svoData(0)
pos1          VAR      svoData(1)
pos2          VAR      svoData(2)
pos3          VAR      svoData(3)
pos4          VAR      svoData(4)
pos5          VAR      svoData(5)
pos6          VAR      svoData(6)
pos7          VAR      svoData(7)
svoTix        VAR      svoData(8)          ' isr divider
svoFrame_LSB  VAR      svoData(9)          ' frame timer
svoFrame_MSB  VAR      svoData(10)
svoIdx        VAR      svoData(11)         ' active servo pointer
svoTimer      VAR      svoData(12)         ' pulse timer
svoPin        VAR      svoData(13)         ' active servo pin

' =====
INTERRUPT NOPRESERVE 307_200          ' run every 3.255 uS
' =====

Mark_ISR:
  ASM
  SETB  isrFlag          ' (1)
  ENDASM

' -----
' RX UART
' -----
'

Receive:
  ASM
  BANK  rxSerial          ' (1)
  JB    rxBufCnt.3, RX_Done ' (2/4) skip if buffer is full
  MOVB  C, RX             ' (4)  sample serial input
  TEST  rxCount           ' (1)  receiving now?
  JNZ   RX_Bit            ' (2/4) yes, get next bit
  MOV   W, #9             ' (1)  no, prep for next byte
  SC                      ' (1/2)
  MOV   rxCount, W        ' (1)  if start, load bit count
  MOV   rxDivide, #Baud1x5 ' (2)  prep for 1.5 bit periods

RX_Bit:
  DJNZ  rxDivide, RX_Done ' (2/4) complete bit cycle?
  MOV   rxDivide, #Baud1x0 ' (2)  yes, reload bit timer
  DEC   rxCount           ' (1)  update bit count
  SZ                      ' (1/2)
  RR    rxByte            ' (1)  position for next bit
  SZ                      ' (1/2)
  JMP   RX_Done           ' (3)

RX_Buffer:
  MOV   W, #rxBuf         ' (1)  point to buffer head
  ADD   W, rxHead         ' (1)
  MOV   FSR, W            ' (1)

```

```

MOV   IND, rxByte           ' (2)  move rxByte to head
INC   rxHead                ' (1)  update head
CLRB  rxHead.3             ' (1)  keep 0..7
INC   rxBufCnt              ' (1)  update buffer count
SETB  rxReady               ' (1)  set ready flag

RX_Done:
BANK  0                     ' (1)
ENDASM

' -----
' Servo Processing
' -----
'
Test_Servo_Tix:
ASM
BANK  svoData                ' (1)
INC   svoTix                 ' (1)  update divider
CJB   svoTix, #3, Servo_Done ' (4/6) done?
CLR   svoTix                 ' (1)  yes, reset for next

' Code below this point runs every 9.766 uS

Check_Frame_Timer:
CJNE  svoFrame_LSB, #2048 & 255, Inc_FrTmr ' (4/6) svoFrame = 2048 (20 ms)?
CJNE  svoFrame_MSB, #2048 >> 8, Inc_FrTmr ' (4/6)
CLR   svoFrame_LSB          ' (1)  yes, reset
CLR   svoFrame_MSB          ' (1)
MOV   svoPin, #%00000001    ' (2)  start servo sequence
CLR   svoIdx                 ' (1)  point to servo 0
MOV   FSR, #pos              ' (2)
MOV   svoTimer, IND         ' (2)
JMP   Refresh_Servo_Outs    ' (3)

Inc_FrTmr:
INC   svoFrame_LSB          ' (1)  INC svoFrame
ADDB  svoFrame_MSB, Z       ' (2)

Check_Servo_Timer:
TEST  svoPin                 ' (1)  any servos on?
SNZ   ' (1)
JMP   Servo_Done            ' (1)  no, exit
DEC   svoTimer               ' (1)  yes, update timer
SZ    ' (1)  still running?
JMP   Servo_Done            ' (1)  yes, exit

Reload_Servo_Timer:
INC   svoIdx                 ' (1)  point to next servo
CLRB  svoidx.3              ' (1)  keep 0 - 7
MOV   W, #pos                ' (1)  get pulse timing
ADD   W, svoIdx              ' (1)
MOV   FSR, W                 ' (1)
MOV   W, IND                 ' (1)
MOV   svoTimer, W           ' (1)  move to timer

Select_Next_Servo:
CLC   ' (1)
RL    svoPin                 ' (1)

Refresh_Servo_Outs:
MOV   ServoCtrl1, svoPin     ' (2)  update outputs

Servo_Done:
BANK  0                     ' (1)
ENDASM

RETURNINT

' =====

```

```

PROGRAM Start
' =====
'
' -----
' Subroutine / Function Declarations
' -----

RX_BYTE          FUNC    1, 0          ' receive a byte

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

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          ' servo channel?
    value = value MIN LO_LIMIT      ' force to legal limits
    value = value MAX HI_LIMIT
    pos(chan) = value
ENDIF

GOTO Main

' -----
' Subroutine / Function Code
' -----

' Use: aByte = RX_BYTE
' -- returns "aByte" from 8-byte circular buffer
' -- will wait if buffer is presently empty
' -- rxBufCnt holds byte count of receive buffer ( 0 to 8 )
' -- rxReady indicates buffer state (0 = empty)

FUNC RX_BYTE
ASM
    BANK    rxSerial
    JNB     rxReady, @RX_BYTE        ' wait if buffer empty
    MOV     W, #rxBuf                ' point to tail
    ADD     W, rxTail
    MOV     FSR, W
    MOV     __PARAM1, IND            ' get byte at tail
    INC     rxTail                    ' update tail
    CLRB   rxTail.3                  ' keep 0..7
    DEC     rxBufCnt                  ' update buffer count
    SNZ                                          ' exit if not zero
    CLRB   rxReady                    ' else clear ready flag
    BANK    0
ENDASM
ENDFUNC

' -----
' User Data
' -----

```

```

=====
'
' File..... VEX_Demod.SXB
' Purpose... Dual-mode servo demodulator for VEX receiver
' Author.... Jon Williams
'           Copyright (c) 2007 Jon Williams
'           Some Rights Reserved
'           -- see http://creativecommons.org/licenses/by/3.0/
' E-mail.... jwilliams@efx-tek.com
' Started...
' Updated... 02 MAR 2007
'
=====
'
' -----
' Program Description
' -----
'
' Simple program to convert PPM stream from VEX RC receiver to servo
' output pulses.
'
' See: http://www.vexfan.com/viewtopic.php?t=227
'
' The program supports two modes: with JP1 removed the program behaves
' like a standard, six-channel servo controller; with JP1 installed,
' channels five and six become on/off digital outputs.
'
' -----
' Conditional Compilation Symbols
' -----
'
' -----
' Device Settings
' -----
'
DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX, BOR42
FREQ            4_000_000
ID              "VexDemod"
'
' -----
' IO Pins
' -----
'
PPM             PIN      RA.0 INPUT          ' PPM in (pull-up w/10k)
MJumper         PIN      RA.1 INPUT PULLUP
'
UndefinedRA2    PIN      RA.2 INPUT PULLUP
UndefinedRA3    PIN      RA.3 INPUT PULLUP
'
UndefinedRB     PIN      RB   INPUT PULLUP
'
SvoPort         PIN      RC
Servo1          PIN      RC.0 OUTPUT
Servo2          PIN      RC.1 OUTPUT
Servo3          PIN      RC.2 OUTPUT
Servo4          PIN      RC.3 OUTPUT
Control1        PIN      RC.4 OUTPUT          ' servo/digital pins
Control2        PIN      RC.5 OUTPUT
'
UndefinedRC6    PIN      RC.6 INPUT  PULLUP
UndefinedRC7    PIN      RC.7 INPUT  PULLUP
'
' -----
' Constants
' -----

```

```

IsOn          CON      1
IsOff         CON      0

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

pulseTmr     VAR      Word
svoPntr      VAR      Byte

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

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

WAIT_SYNC    SUB      0          ' wait for sync pulse
WAIT_HI_LO   SUB      0          ' wait for 1-to-0 on PPM
WAIT_LO_HI   SUB      0          ' wait for 0-to-1 on PPM

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

Start:
  SvoPort = %00000000          ' clear servos

Main:
  IF MJumper = 0 THEN Servo_Plus          ' check mode jumper

' -----
' Standard servo control
' -----
'
Standard_Servos:
  WAIT_SYNC
  svoPntr = %0000_0001
  DO
    SvoPort = svoPntr          ' activate servo pin
    WAIT_HI_LO          ' let framing pulse finish
    WAIT_LO_HI          ' let timing pulse finish
    svoPntr = svoPntr << 1          ' point to next
  LOOP UNTIL svoPntr = %0100_0000
  GOTO Start

' -----
' Four servos + two on/off
' -----
'
Servo_Plus:
  WAIT_SYNC
  svoPntr = %0000_0001
  DO
    SvoPort = SvoPort | svoPntr          ' activate servo pin
    WAIT_HI_LO          ' let framing pulse finish
    WAIT_LO_HI          ' let timing pulse finish
    svoPntr = svoPntr << 1          ' point to next
    SvoPort = SvoPort & %0011_0000          ' protect control pins
  LOOP UNTIL svoPntr = %0001_0000

Ctrl_Port1:

```

```

WAIT_HI_LO                                ' let framing pulse finish
pulseTmr = 0                               ' reset pulse timer
DO WHILE PPM = 0                           ' measure low-going pulse
  PAUSEUS 10
  INC pulseTmr
LOOP

IF pulseTmr < 60 THEN                      ' top button pressed?
  Control1 = IsOn
ELSEIF pulseTmr > 110 THEN                ' bottom button pressed
  Control1 = IsOff
ENDIF

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

IF pulseTmr < 60 THEN                      ' top button pressed?
  Control2 = IsOn
ELSEIF pulseTmr > 110 THEN                ' bottom button pressed
  Control2 = IsOff
ENDIF

GOTO Main

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

' Use: WAIT_SYNC
' -- waits for sync spacing in PPM stream
' -- measured pulse is 8.9 mS; this routine waits for 4+ mS pulse

SUB WAIT_SYNC
  pulseTmr = 0                             ' reset pulse timer
  DO WHILE PPM = 0                         ' measure low-going pulse
    PAUSEUS 10
    INC pulseTmr
  LOOP
  IF pulseTmr < 400 THEN WAIT_SYNC          ' hold for valid sync
ENDSUB

' -----

' Use: WAIT_LO_HI
' --waits for low-to-high transistion on PPM line

SUB WAIT_LO_HI
  DO WHILE PPM = 0
  LOOP
ENDSUB

' -----

' Use: WAIT_HI_LO
' --waits for high-to-low transistion on PPM line

SUB WAIT_HI_LO
  DO WHILE PPM = 1
  LOOP
ENDSUB

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

```