



Column #144, July 2007 by Jon Williams:

## Rev It Up

*For me, one of the most exciting aspects of working with the SX and programming it in SX/B is the ability to create accessory devices like the animatronics controller. With what we've learned it seems that the next logical step is a Parallax AppMod-compatible device; since Parallax already has a great servo controller (the PSC) that uses this protocol, let's build a compatible motor driver. And for those just getting started or don't want to have a PCB etched, have no fear, we're going to whip this baby together with a handful of parts and Parallax's incredibly low-priced SX28 proto board.*

Some of you probably remember an article last year where we experimented with interrupts and motor control; well, that was just scratching the surface. We've all come a long way since then, as has SX/B, and we're going to put what we've learned to work. Since the AppMod protocol uses a single wire for communications we'll have to deal with that, and while we're at it we'll implement a baud selection input so that the device can be used with the BASIC Stamp 1 at 2400 baud, or with the BASIC Stamp 2 family at 38.4k baud. What you'll soon see is that the simplest aspect is implementing the PWM signals that control the motor outputs (we'll control up to two motors with this device – and expanding is pretty easy).

As ever, we should start with the specifications. If we're going to follow the AppMod protocol we should implement a command to return the firmware version of the device. The message is simple:

```
"!MC" {address} "V"
```

**The Nuts & Volts of BASIC Stamps 2007**

## Column #144: Rev It Up

Of course, the quotes aren't actually used in the message, they're used in the line above to indicate string or ASCII character values versus numbers. The device will use a two-bit address, so valid addresses are from %00 (board 0) to %11 (board 3).

The return value for the version request is a simple three-byte string that holds the major and minor version numbers, separated by a period. Like this:

```
"0.1"
```

Again, this value is returned as a string and conversion may be required for some applications of the version number.

Now that you've got an idea of how the AppMod protocol works, here are the other commands that we'll implement:

- "G" – get device status, returns three bytes (speed and direction bits)
- "S" – sets the speed and direction of one or both motors, nothing returned
- "X" – stops both motors if running, nothing returned

This is a very simple command set, and as you'll see there's a lot of room in the code for additional features if you choose to add them for your particular projects.

Like the last couple projects, this one uses the ISR to handle a lot of the tough work "in the background," including delay timing, receive and transmit UARTs, and the PWM output for the motors. You'll see that the UART code is identical to last time, the exception being that the bit delay values are loaded from variables so that we can select between the supported baud rates of 2400 and 38.4k.

Here's a little secret – a cheat if you will – that's used to keep the transmitter UART code simple: even though we're supposed to be using open-true communications where a transmitted "1" will pull the serial line low and a transmitted "0" will allow the serial line to be released to the pull-up, this is a bit of a headache; so... what we're going to do is put a resistor inline with the serial pin to protect connected devices in the event of an electrical conflict which, in fact, is not likely to happen as we'll mostly be using a BASIC Stamp which does support open-true communications.

Now, that said, we do need to check the state of the transmit UART before putting the device into receive mode; that's easily done by checking the variable called *txCount* that keeps track of the bits being sent. In addition to the transmitter check we'll also incorporate a baud

selection routine and the ability to convert lower-case letters to upper-case when needed by the command processor; this last section simplifies the mainline code.

```

FUNC RX_BYTE
  ASM
  BANK serial
  TEST txCount
  JNZ RX_BYTE
  BANK 0
  ENDASM

  IF __PARAMCNT = 1 THEN
    tmpB2 = __PARAM1.0
  ELSE
    tmpB2 = 0
  ENDIF

  INPUT Sio
  IF BaudRate = BR2400 THEN
    baud1x0 = BaudSlow1x0
    baud1x5 = BaudSlow1x5
  ELSE
    baud1x0 = BaudFast1x0
    baud1x5 = BaudFast1x5
  ENDIF

  DO WHILE rxReady = No
  LOOP
  tmpB1 = rxByte
  rxReady = No

  IF tmpB2.0 = ToUpper THEN
    IF tmpB1 >= "a" THEN
      IF tmpB1 <= "z" THEN
        tmpB1.5 = 0
      ENDIF
    ENDIF
  ENDIF

  RETURN tmpB1
ENDFUNC

```

On entry to the RX\_BYTE function the value of *txCount* is tested to see if it's zero (not transmitting); the code will jump back to the start of RX\_BYTE until anything currently being transmitted is finished. With the transmit buffer clear we'll check to see if a parameter was passed; this will contain a flag (in bit 0 of the parameter) that will indicate the desire to convert a lowercase letter to uppercase, all other values will pass through unmodified.

## Column #144: Rev It Up

Now it's safe to make the serial line an input and then check the baud rate jumper on the controller. The setting of the jumper handles the bit count values using for the start and data bit timing. At this point it's a waiting game for the receive flag, *rxReady*, to go from zero to one. Once a byte comes in we check to see if case conversion was specified and if so if the received byte is a lowercase letter. Conversion to uppercase is a simple matter of clearing bit 5 of the byte (the same as subtracting \$20).

The transmit subroutine is a little simpler, yet also needs to check the state of the transmit buffer so that a character doesn't overrun the one currently being transmitted.

```
SUB TX_BYTE
  ASM
  BANK serial
  TEST txCount
  JNZ TX_BYTE
  BANK 0
  ENDASM

  IF BaudRate = BR2400 THEN
    baud1x0 = BaudSlow1x0
  ELSE
    baud1x0 = BaudFast1x0
  ENDIF
  HIGH Sio

  ASM
  BANK serial
  MOV txHi, __PARAM1
  CLR txLo
  MOV txCount, #11
  BANK 0
  ENDASM
ENDSUB
```

When all is clear to transmit the serial line is set high – this makes it an output and puts it into the idle state. One final note on transmitting: occasionally SX/B users will ask how to add extra stop bits while transmitting. When using this ISR-based transmit UART it's easy; all you have to do is change the value of *txCount* (which also starts the UART). Normally, a value of 10 would be used for one start bit, eight data bits, and one stop bit. I chose to use 11 in this program because I will be using the BS2 most of the time and the extra stop bit gives it just a little extra time between bytes.

Alright, now that we can receive bytes from and send bytes to our host, let's look at the main loop of the program that handles host commands and requests.

## The Nuts & Volts of BASIC Stamps 2007

To begin we will monitor the serial stream for the device's header: "!MC." In the BASIC Stamp we have a WAIT modifier for SERIN that doesn't exist in SX/B. That's not a problem because the functionality is very easy to code:

```
Main:
char = RX_BYTE
IF char <> "!" THEN Main
char = RX_BYTE ToUpper
IF char <> "M" THEN Main
char = RX_BYTE ToUpper
IF char <> "C" THEN Main
```

Yes, it really is that simple – and as you can see uppercase conversion when needed keeps things nice and tidy. Here's what that section would look like without the uppercase conversion built into RX\_BYTE:

```
Main:
char = RX_BYTE
IF char <> "!" THEN Main

Header2:
char = RX_BYTE
IF char = "m" THEN Header3
IF char = "M" THEN Header3
GOTO Main

Header3:
char = RX_BYTE
IF char = "c" THEN Check_Addr
IF char = "C" THEN Check_Addr
GOTO Main
```

I think you'll agree that putting the case conversion code into the RX\_BYTE function is worth the effort, especially when we have a program waiting on a long string that could be upper- or lowercase

The next step in the process is to check the board address. This architecture also allows for a "global" address of 255 that affects all boards; this is especially useful for shutting down all boards with a single command from the host:

```
Check_Addr:
char = RX_BYTE
IF char = CmdAll THEN
    allCall = Yes
    GOTO Get_Cmd
ELSE
```

## Column #144: Rev It Up

```
    allCall = No
ENDIF
addr = %00
addr.0 = ~A0
addr.1 = ~A1
IF char <> addr THEN Main
```

If the received address is not global (255) then it gets compared against the jumpers. Note that we're using the internal pull-ups for these pins so the code inverts the bits to make them active high. As with the header a mismatch causes the program to jump back to Main and wait for the next command.

Next up is the command processor.

```
Get_Cmd:
char = RX_BYTE ToUpper
IF char = "V" THEN Show_Version
IF char = "G" THEN Get_Status
IF char = "S" THEN Set_Motor
IF char = "X" THEN Stop_Motors
GOTO Main
```

Again, dirt simple which is always best in my book, especially when writing modular code that can be used across a variety of projects as this AppMod framework can be. In PBASIC we would use ON..GOTO or BRANCH, but in SX/B I find the IF..THEN approach cleaner. Remember that this code gets compiled to pure assembly so the resulting output is very fast – this structure is deceptively efficient.

```
Get_Cmd:
char = RX_BYTE ToUpper
IF char = "V" THEN Show_Version
IF char = "G" THEN Get_Status
IF char = "S" THEN Set_Motor
IF char = "X" THEN Stop_Motors
GOTO Main
```

Since the Version request uses another helpful subroutine we'll have a look at that.

```
Show_Version:
IF allCall = Yes THEN Main
DELAY_MS 15
TX_STR Rev_Code
GOTO Main
```

## The Nuts & Volts of BASIC Stamps 2007

Note that we check to see if the command sent contained the global address; when this is the case we don't want to respond because it could cause conflict with other devices on the line. When the command is intended for this board we will pause just a bit to let the host (like a BASIC Stamp 1) get ready for the serial input. After that we can send the firmware version which is stored as a z-string in a DATA statement. We pass the address (resolved by the compiler from the label) to TX\_STR.

```
SUB TX_STR
  tmpW1 = __WPARAM12
  DO
    READINC tmpW1, tmpB1
    IF tmpB1 = 0 THEN EXIT
    TX_BYTE tmpB1
  LOOP
ENDSUB
```

What makes this subroutine really tick is READINC, a fairly new addition to SX/B that will read a byte from the specified address pointer and then automatically increment that pointer. That it works with words also keeps the subroutine very clean. This routine is in my "standard" set and I suggest it be in yours as well.

As this project is a motor controller, let's have a look at the command that sets speed, "S." The byte the follows this command will be zero (all motors), or 1 or 2 for a specific motor. Here's the code:

```
Set_Motor:
  char = RX_BYTE
  IF char = 0 THEN
    speed1 = RX_BYTE
    tmpB1 = RX_BYTE
    dir1 = tmpB1.0
    speed2 = RX_BYTE
    tmpB1 = RX_BYTE
    dir2 = tmpB1.0
  ELSEIF char = 1 THEN
    speed1 = RX_BYTE
    tmpB1 = RX_BYTE
    dir1 = tmpB1.0
  ELSEIF char = 2 THEN
    speed2 = RX_BYTE
    tmpB1 = RX_BYTE
    dir2 = tmpB1.0
  ENDIF
  GOTO Main
```

## Column #144: Rev It Up

The use of IF-THEN-ELSEIF allows us to check for valid values without defaulting to something that could be incorrect; look closely, there is no ELSE clause in the structure. For example, if a motor number of three were specified all of the IF clauses would fail and neither motor would be affected. For each motor we expect to receive speed (0 for 0%, 255 for 100%) and direction (0 for forward, 1 for reverse). The direction for each motor is encoded in bit flags so we will copy those bits from the associated bytes in the stream command stream.

### Get Your Motors Runnin'

Now that we have speed and direction values in the controller, how do the motors get moved? It's actually pretty simple with a bit of accumulator-based PWM. The program keeps two values for running the motor at a given duty cycle: the speed sent by the host and an accumulator that is used to control the outputs.

On each pass through the interrupt a variable called *mTimer* is incremented; this is the timer that controls motor updates. When this value rolls over to zero we will reload the motor accumulators with the speed values sent by the host. At this point the motor outputs are both set to off.

Then, for each motor, the accumulator is incremented and when it rolls over to zero the correct output (set by motor direction) is made high. So you see, the higher the motor speed setting, the more quickly the rollover to zero will happen and the longer the motor will be on as a percentage of the cycle, hence will run faster. This is a neat, very efficient trick and is even easier when it's just one output to be controlled as with LED brightness, or charging an RC circuit to create an analog output.

```

Check_Motors:
  ASM
  BANK  motor
  IJNZ  mTimer, M1_Fwd
  MOV   m1Acc, speed1
  MOV   m2Acc, speed2
  MOV   MotorCtrl, #%0000

M1_Fwd:
  IJNZ  m1Acc, M2_Fwd
  JB    dir1, M1_Rev
  SETB  M1_A
  JMP   M2_Fwd

M1_Rev:
  SETB  M1_B

M2_Fwd:
  IJNZ  m2Acc, Motors_Done
  JB    dir2, M2_Rev
  SETB  M2_A
  JMP   Motors_Done

M2_Rev:
  SETB  M2_B

Motors_Done:
  BANK  0
  ENDASM

```

### Construction Notes

This one is easy thanks to our friends at Parallax: I used the \$10 SX28 Proto Board and components that most of you have in your supply bins – with the possible exception of the L293D. Figure 144.1 shows the schematic and connections to the SX28 Proto Board and Figure 144.2 will give you an idea of what it looks like when wired up and running. Note the addition of TB1 which allows us to connect Vin to V-Motor if we want to run from the SX28 Proto Board Supply (the power switch needs to be in position 2 for this). If separate supply is required this should be connected between V-Motor (TB2.1) and Ground (TB2.6).

Column #144: Rev It Up

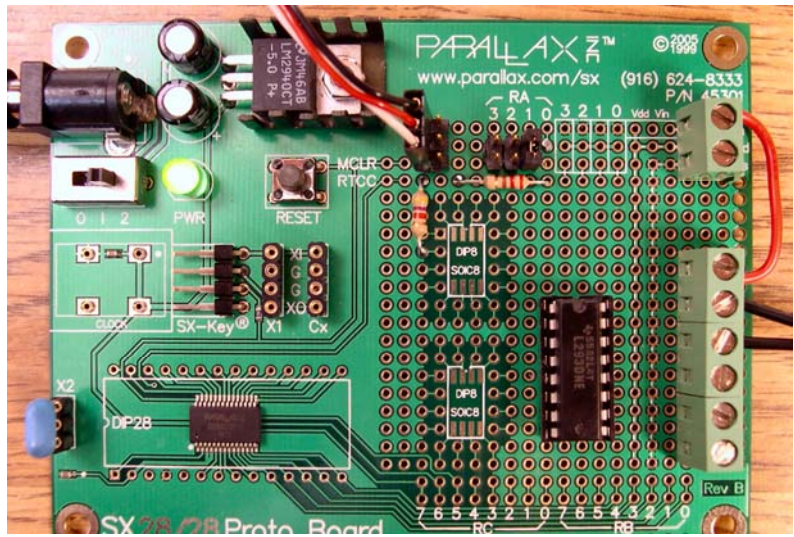
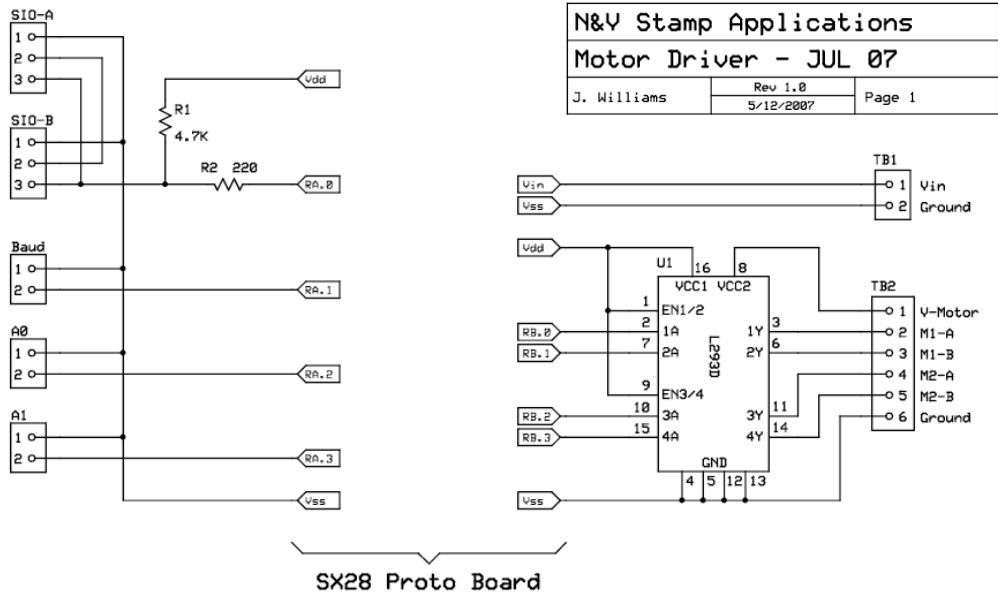


Figure 144.1: Motor Driver Schematic (top), Figure 144.2: On SX Proto Board (bottom)

### **Getting Started with the SX**

The last few articles have stirred up quite a lot of interest from those using other processors – and why not?, the SX is lean, mean, and darned cool, and with SX/B there's a lot of power to be had without a gigantic cash outlay. Case in point: the SX28 proto board that I used for this project. There's not a lot of places you can get a fully-built microcontroller board with a beefy power supply and nice breadboarding space for ten bucks. They're perfect for one-off projects, and can be real life-savers when you're pressed for time.

Just last summer I met with a client who wanted a custom solenoid driver for a special project. We met on a Friday afternoon, discussed the project, and reasonably concluded that it would take about two weeks to complete. Then, just as I was about to head home the client asked if I could whip up something "quick and dirty" for Monday. "Are you kidding?" I thought... but my actual answer was, "Let me see what I can do."

In the end, of course, I hand-built a prototype using the SX28 proto board and used manual inputs (potentiometers) instead of the fancy digital display originally called for in the design. Well, the SX28 proto board (and SX/B) got me out of a bind as I was able to get the "quick and dirty" version going, but it also cost me a few bucks as the "quick" version worked so well that the client cancelled the rest of the contract. I guess the lesson is to keep the SX28 proto board handy for emergencies, just don't tell your customers that Parallax has done a lot of the hard work for you!

All kidding aside, the proto boards (there's one for the SX48 as well) are probably the least expensive way to started with the SX – after you get a programming tool. For programming you have two choices: the SX-Key and the SX-Blitz. I always recommend the Key because it lets you tap into all the power of the IDE, including debugging. The Blitz is purely a programming device; if you're really on a budget then this will work. Many Blitz users have found Guenther Daubach's SX-Sim a great tool for stepping through code when things aren't working quite right. If you can save a few pennies for the SX-Key I can promise you it will be worth it.

### **SX/B 2.0?**

You may have noticed a slightly different format to subroutines and functions: specifically, they begin with SUB or FUNC, and end with ENDSUB or ENDFUNC. This is not a requirement of the current compiler (version 1.51.03) but it is good habit to get into so that your code will be compatible with SX/B version 2.0 which is scheduled to be released later this year. I'm told it will include local variables and is designed to make multitasking programs easier to create – it should be very cool!

### **Animatronics Controller Update**

The May article generated an enormous amount of feedback – even one piece of quasi “hate mail” (thank you, anonymous e-mailer, for making me laugh out loud). I think showing a framework for servo control is really exciting a lot of folks as we use them in so many places (e.g., robotics).

There are two issues I want to address: First, I made a mistake in the regulator I specified – forgive me a small bit of laziness; I had used that regulator (LF50CP) is a previous project but neglected to go back and check its specs for this one. As one kind reader pointed out, that regulator is only good 0.5A and that might not be sufficient when running up eight servos as the board is designed to do. If you’re planning to build that board, use the LM2940-5.0 (Digi-Key LM2940CT-5.0-ND) or the L4940V5 (Mouser 511- L4940V5) and put a heat sink on it if you’re going to control more than a few servos.

The other issue I want to address is device control. In May I talked about developing a program that would let me synchronize the servo and digital outputs with an audio track – this generated more mail than most columns and I appreciate all the great suggestions. That said, I’ve put my personal development plans on hold.

What?... You see, in doing research on AC lamp dimming (something we’ll work with in November) I came across this really neat piece of freeware called Vixen (Figure 144.3) and my new best friend is Vixen’s developer, KC Oaks. Vixen is tremendously popular with amateur – and I dare say professional – Christmas lighting enthusiasts. Vixen does what I need: it allows me to synchronize outputs to an audio track (and that’s just scratching the surface).

One of Vixen’s many strengths is its open architecture. KC designed the program so that custom hardware drivers could be written and installed allowing users to control just about anything – including my animatronics controller. I speculated that I could convert the 0% to 100% lighting output on eight channels to servo position and send the packet like I did with the little VB program. I contacted KC, he whipped up a driver and *cha-ching*, it works! After working with it a bit we added some refinements as shown by the driver setup dialog in Figure 144.4.

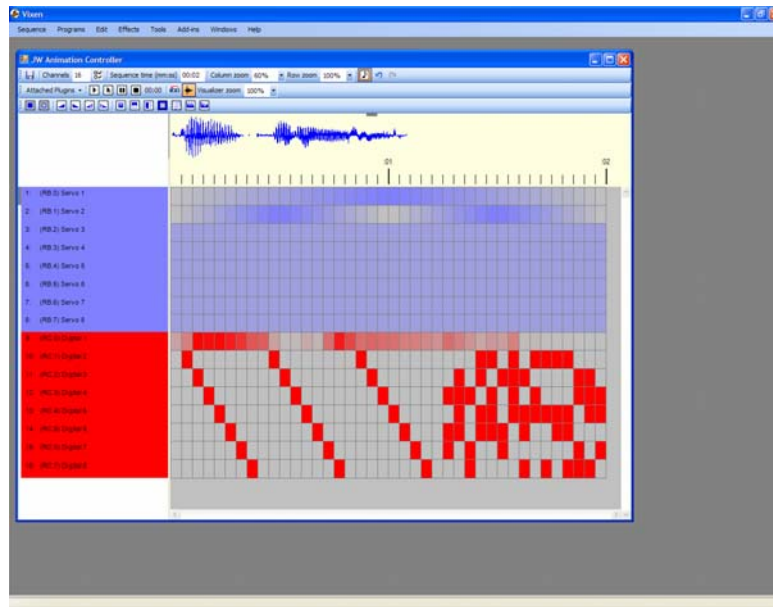


Figure 144.3: The Vixen Software

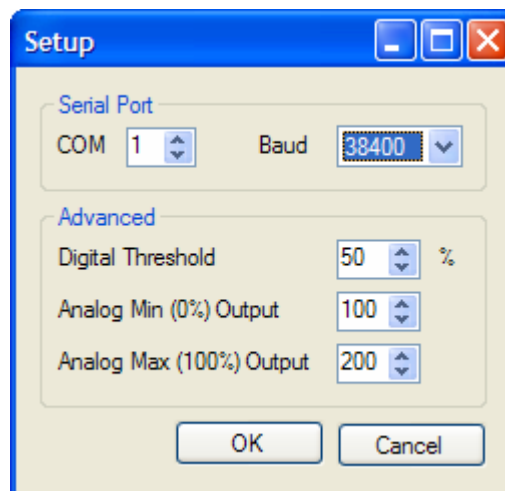


Figure 144.4: The Driver Setup Dialog

## Column #144: Rev It Up

The JWAC (Jon Williams Animatronics Controller) setup dialog lets us select the serial port, baud rate (in case you change your SX firmware) and control settings for the servo (analog) and digital outputs. The threshold value is what gets used to convert the levels in the digital channels (8 – 16) to an on-off bit. You could, for example, use the Waveform add-in to set event levels on a channel such that they follow the envelope of your audio track. With the threshold setting you could determine the audio level that turns on an output, causing the output to flash in sync with audio volume changes.

The analog levels allow you to adjust for servo output range; the default values of 100 and 200 correspond to standard servo control pulse values of 1.0 and 2.0 milliseconds; the minimum setting lets you adjust that end down to 50 (0.5 ms) and the maximum setting lets you adjust that end up to 250 (2.5 ms) – this should provide the flexibility required for a wide range of servos.

Be sure to visit the Vixen web site for updates, add-ins, and to participate in their [very active] user forums. And for those of you who have the USB version of the PSC (Parallax Servo Controller), there is a driver in the works for it as well.

And just to see how much fun a PC-driven animatronics project can be, pop on over to Peter Montgomery's web site, [www.socalhalloween.com](http://www.socalhalloween.com), to find out. Peter created custom software and controllers (the servo controller uses an SX28) to run his display, and yet there's no reason you can create something similar with Vixen, a bit of skill and artistic talent, and a whole lot of patience. I had the good fortune of seeing Peter's display on Halloween night last year and it was another major inspiration for designing my animatronics controller and ultimately finding Vixen to control it.

Well, have fun the new motor controller and with Vixen, too. Until next time, Happy Stamping!

### Resources

Jon Williams  
[jwilliams@efx-tek.com](mailto:jwilliams@efx-tek.com)

Parallax, Inc.  
[www.parallax.com](http://www.parallax.com)

Vixen  
[www.vixenlights.com](http://www.vixenlights.com)

**The Nuts & Volts of BASIC Stamps 2007**

Project Bill of Materials		
Designator	Value	Source
PCB	SX28 Proto Board	Parallax 45302
RES	20 MHz resonator	Parallax 250-02060
R1	4.7K	Mouser 80-C315C104M5U
R2	220	Mouser 299-220-RC
TB1	2-pos term	Mouser 571-2828362
TB2	2-pos term (3x)	Mouser 571-2828362
UI	L293D	Mouser 511-L293D
	Socket for U1	Mouser 535-16-3518-10
X1-X5	0.1 pin header	Mouser 517-6111TG
	Jumpers for X2-X5	Mouser 525-ME-100

### Source Code

```

' =====
'
' File..... MC-2_Motor_Controller.SXB
' Purpose... Parallax AppMod-compatible dual motor controller
' Author.... Jon Williams, EFX-TEK
'           Copyright (c) 2007 EFX-TEK
'           Some Rights Reserved
'           -- see http://creativecommons.org/licenses/by/3.0/
'           -- ISR UART code originally by Chip Gracey
' E-mail.... jwilliams@efx-tek.com
' Started...
' Updated... 13 MAY 2007
'
' =====
'
' -----
' Program Description
' -----
'
' MC-2 Dual Motor Controller (AppMod protocol compatible)
'
' Commands:
'
'   "!MC", addr, "V"           - Returns version string (3 byte)
'   "!MC", addr, "G"           - Returns status (3 bytes)
'   "!MC", addr, "S", mtr, data - set motor speed (one or both)
'   "!MC", addr, "X"           - Stop both motors

```

## Column #144: Rev It Up

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

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

DEVICE          SX28, OSCXT2, TURBO, STACKX, OPTIONX, BOR42
FREQ            20_000_000
ID              "MC2 v0.1"

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

Sio              PIN      RA.0  INPUT      ' serial I/O to host
BaudRate        PIN      RA.1  INPUT  PULLUP    ' B/R select input
A1              PIN      RA.2  INPUT  PULLUP    ' address jumpers (%00-%11)
A0              PIN      RA.3  INPUT  PULLUP

MotorCtrl       PIN      RB
M1_A            PIN      RB.0  OUTPUT    ' L293D control outputs
M1_B            PIN      RB.1  OUTPUT
M2_A            PIN      RB.2  OUTPUT
M2_B            PIN      RB.3  OUTPUT

UnusedRB4       PIN      RB.4  PULLUP
UnusedRB5       PIN      RB.5  PULLUP
UnusedRB6       PIN      RB.6  PULLUP
UnusedRB7       PIN      RB.7  PULLUP

UnusedRC        PIN      RC    PULLUP

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

CmdAll          CON      $FF      ' command all devices

Yes             CON      1
No              CON      0

MFwd            CON      0
MRev            CON      1

ToUpper        CON      1      ' for serial in commands
```

## The Nuts & Volts of BASIC Stamps 2007

```

BR2400      CON      1          ' no B/R jumper
BR38K4      CON      0          ' B/R jumper installed

' Dividers for ISR UART
' -- values set for 6.51 uS interrupt rate

Baud2400    CON      64
Baud4800    CON      32
Baud9600    CON      16
Baud19K2    CON      8
Baud38K4    CON      4

BaudSlow1x0 CON      Baud2400
BaudSlow1x5 CON      Baud2400 * 3 / 2

BaudFast1x0 CON      Baud38K4
BaudFast1x5 CON      Baud38K4 * 3 / 2

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

flags       VAR      Byte
rxReady     VAR      flags.0          ' serial byte available
allCall     VAR      flags.1          ' request for all boards

msTix       VAR      Byte          ' for ISR ms delays

char        VAR      Byte          ' character in/out
addr        VAR      Byte          ' board address (%00-%11)

tmpB1       VAR      Byte          ' sub/func work vars
tmpB2       VAR      Byte
tmpW1       VAR      Word

serial      VAR      Byte (16)      ' bank for serial vars
txCount     VAR      serial(0)
txDivide    VAR      serial(1)
txLo        VAR      serial(2)
txHi        VAR      serial(3)
rxCount     VAR      serial(4)
rxDivide    VAR      serial(5)
rxByte      VAR      serial(6)
baud1x0     VAR      serial(7)
baud1x5     VAR      serial(8)

motor       VAR      Byte (16)      ' bank for motor control
mFlags      VAR      motor(0)

```

## Column #144: Rev It Up

```
dir1          VAR    mFlags.0
dir2          VAR    mFlags.1
mTimer        VAR    motor(1)      ' pwm clock
speed1        VAR    motor(2)      ' current motor setting
speed2        VAR    motor(3)
m1Acc         VAR    motor(4)      ' pwm accumulators
m2Acc         VAR    motor(5)

' -----
' INTERRUPT NOCODE 153_600          ' every 6.51 uS
' -----

MS_Timer:
  ASM
  BANK msTix
  CJE msTix, #0, Timer_Done        ' timer running?
  DEC msTix                          ' yes, decrement

Timer_Done:
  BANK 0
  ENDASM

Receive:
  ASM
  BANK flags
  JB rxReady, RX_Done              ' skip if byte waiting
  BANK serial
  MOVB C, Sio                       ' sample serial input
  TEST rxCount                      ' receiving now?
  JNZ RX_Bit                        ' yes, get next bit
  MOV W, #9                          ' no, prep for next byte
  SC
  MOV rxCount, W                    ' if start, load bit count
  MOV rxDivide, baud1x5              ' prep for 1.5 bit periods

RX_Bit:
  DJNZ rxDivide, RX_Done            ' complete bit cycle?
  MOV rxDivide, baud1x0              ' yes, reload bit timer
  DEC rxCount                        ' update bit count
  SZ
  RR rxByte                          ' position for next bit
  JNZ RX_Done
  BANK flags
  SETB rxReady                       ' alert foreground

RX_Done:
  BANK 0
  ENDASM
```

## The Nuts & Volts of BASIC Stamps 2007

```

Transmit:
  ASM
  BANK serial
  TEST txCount          ' transmitting now?
  JZ TX_Done            ' if txCount = 0, no
  DEC txDivide          ' update bit timer
  JNZ TX_Done           ' time for new bit?
  MOV txDivide, baud1x0 ' yes, reload timer
  STC                   ' set for stop bit
  RR txHi               ' rotate TX buf
  RR txLo
  DEC txCount           ' update the bit count
  MOVB Sio, txLo.6     ' output the bit

TX_Done:
  BANK 0
  ENDASM

Check_Motors:
  ASM
  BANK motor
  IJNZ mTimer, M1_Fwd  ' update pwm timer
  MOV m1Acc, speed1   ' reset accumulators
  MOV m2Acc, speed2
  MOV MotorCtrl, #0000 ' clear control outputs

M1_Fwd:
  IJNZ m1Acc, M2_Fwd  ' update pwm accumulator
  JB dir1, M1_Rev     ' check direction
  SETB M1_A           ' enable forward
  JMP M2_Fwd

M1_Rev:
  SETB M1_B           ' enable reverse

M2_Fwd:
  IJNZ m2Acc, Motors_Done
  JB dir2, M2_Rev
  SETB M2_A
  JMP Motors_Done

M2_Rev:
  SETB M2_B

Motors_Done:
  BANK 0
  ENDASM

```

## Column #144: Rev It Up

```
ISR_Exit:
  RETURNINT

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

' -----
' Subroutine / Function Declarations
' -----

DELAY_MS      SUB    1, 2          ' delay in milliseconds
TX_BYTE       SUB    1            ' transmit byte
TX_STR        SUB    2            ' transmit string

RX_BYTE       FUNC   1, 0, 1      ' receive a byte

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

Start:
  GOTO Stop_Motors

Main:
  char = RX_BYTE          ' wait for header
  IF char <> "!" THEN Main
  char = RX_BYTE ToUpper
  IF char <> "M" THEN Main
  char = RX_BYTE ToUpper
  IF char <> "C" THEN Main

Check_Addr:
  char = RX_BYTE          ' rx addr
  IF char = CmdAll THEN  ' all call requested?
    allCall = Yes        ' yes, set flag
    GOTO Get_Cmd         ' and skip ID check
  ELSE
    allCall = No         ' clear global flag
  ENDIF
  addr = %00             ' clear old setting
  addr.0 = ~A0           ' get low bit of address
  addr.1 = ~A1           ' get high bit of address
  IF char <> addr THEN Main ' validate
```

```

Get_Cmd:
  char = RX_BYTE ToUpper          ' rx command byte
  IF char = "V" THEN Show_Version
  IF char = "G" THEN Get_Status
  IF char = "S" THEN Set_Motor
  IF char = "X" THEN Stop_Motors
  GOTO Main

Show_Version:
  IF allCall = Yes THEN Main      ' cmd = "V"
  DELAY_MS 15                     ' prevent response collision
  TX_STR Rev_Code                 ' let host get ready
  GOTO Main

Get_Status:
  IF allCall = Yes THEN Main      ' cmd = "G"
  DELAY_MS 15                     ' prevent response collision
  tmpB1 = dir1                   ' let host get ready
  tmpB1.1 = dir2
  TX_BYTE speed1                 ' speed #1
  TX_BYTE speed2                 ' speed #2
  TX_BYTE tmpB1                  ' direction bits
  GOTO Main

Set_Motor:
  char = RX_BYTE                  ' cmd = "S"
  IF char = 0 THEN
    speed1 = RX_BYTE
    tmpB1 = RX_BYTE
    dir1 = tmpB1.0
    speed2 = RX_BYTE
    tmpB1 = RX_BYTE
    dir2 = tmpB1.0
  ELSEIF char = 1 THEN
    speed1 = RX_BYTE
    tmpB1 = RX_BYTE
    dir1 = tmpB1.0
  ELSEIF char = 2 THEN
    speed2 = RX_BYTE
    tmpB1 = RX_BYTE
    dir2 = tmpB1.0
  ENDIF
  GOTO Main

Stop_Motors:
  speed1 = 0                      ' cmd = "X"
  dir1 = MFwd

```

## Column #144: Rev It Up

```
speed2 = 0
dir2 = MFwd
GOTO Main

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

' Use: DELAY_MS mSecs

SUB DELAY_MS
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1           ' get byte parameter
  ELSE
    tmpW1 = __WPARAM12        ' get word parameter
  ENDIF
  DO WHILE tmpW1 > 0
    msTix = 153                ' set for ~1 ms
    DO
      LOOP UNTIL msTix = 0     ' wait for timer to expire
      DEC tmpW1
    LOOP
  ENDSUB

' -----

' Use: TX_BYTE aByte
' -- transmits one byte at current baud jumper setting

SUB TX_BYTE
  ASM
  BANK serial
  TEST txCount                 ' transmitting now?
  JNZ TX_BYTE                  ' if yes, wait until done
  BANK 0
  ENDASM

  IF BaudRate = BR2400 THEN
    baud1x0 = BaudSlow1x0
  ELSE
    baud1x0 = BaudFast1x0
  ENDIF
  HIGH Sio                     ' set line idle

  ASM
  BANK serial
  MOV txHi, __PARAM1
  CLR txLo
  MOV txCount, #11             ' start + 8 data + 2 stop
  BANK 0
```

## The Nuts & Volts of BASIC Stamps 2007

```

ENDASM
ENDSUB

' -----
' Use: TX_STR [String | Label]
' -- pass embedded string literal or DATA label (zString at label)

SUB TX_STR
  tmpW1 = __WPARAM12           ' get offset/base
  DO
    READINC tmpW1, tmpB1       ' read a character
    IF tmpB1 = 0 THEN EXIT     ' if 0, string complete
    TX_BYTE tmpB1              ' send the byte
  LOOP
ENDSUB

' -----
' Use: result = RX_BYTE { MakeUppercase }
' -- receives one byte at current baud jumper setting
' -- waits for any TX action to finish
' -- will return to caller after byte received
' -- converts alpha to uppercase if [optional] MakeUppercase.0 = 1

FUNC RX_BYTE
  ASM
  BANK serial
  TEST txCount                 ' transmitting now?
  JNZ  RX_BYTE                 ' if yes, wait until done
  BANK 0
  ENDASM

  IF __PARAMCNT = 1 THEN      ' option specified
    tmpB2 = __PARAM1.0       ' yes, save it
  ELSE
    tmpB2 = 0                 ' no, set to default
  ENDIF

  INPUT Sio                    ' prep for RX
  IF BaudRate = BR2400 THEN
    baud1x0 = BaudSlow1x0
    baud1x5 = BaudSlow1x5
  ELSE
    baud1x0 = BaudFast1x0
    baud1x5 = BaudFast1x5
  ENDIF

  DO WHILE rxReady = No      ' wait for character
  LOOP
  tmpB1 = rxByte              ' get from buffer

```

## Column #144: Rev It Up

```
rxReady = No                                ' allow new RX

IF tmpB2.0 = ToUpper THEN                    ' convert to uppercase?
  IF tmpB1 >= "a" THEN                        ' lowercase?
    IF tmpB1 <= "z" THEN
      tmpB1.5 = 0                             ' ...yes, make uppercase
    ENDIF
  ENDIF
ENDIF

RETURN tmpB1                                ' send char back to caller
ENDFUNC

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

Rev_Code:
DATA "0.1", 0
```

```
' =====
'
' File..... MC2_Test.BS2
' Purpose... Test program for MC-2 Motor Controller
' Author.... Jon Williams, EFX-TEK
' E-mail.... jwilliams@efx-tek.com
' Started...
' Updated... 17 MAY 2007
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =====
'
' -----[ Program Description ]-----
'
' Simple test program for the MC-2 motor controller project. Will work
' at 2400 (BS1-compatible) or 38.4k baud.
'
' -----[ Revision History ]-----
'
' -----[ I/O Definitions ]-----
'
Sio          PIN      15
```

## The Nuts & Volts of BASIC Stamps 2007

```

' -----[ Constants ]-----
#SELECT $STAMP
#CASE BS2, BS2E, BS2PE
  T2400      CON    396
  T38K4      CON    6
#CASE BS2SX, BS2P
  T2400      CON   1021
  T38K4      CON    45
#CASE BS2PX
  T2400      CON   1646
  T38K4      CON    84
#ENDSELECT

Open         CON    $8000
Baud         CON    Open + T38K4

Addr         CON    %00          ' %00 - %11

MFwd         CON    0            ' motor forward
MRev         CON    1            ' motor reverse

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

id           VAR    Byte (3)     ' board rev code
speed1       VAR    Byte         ' motor speed
speed2       VAR    Byte
status       VAR    Byte         ' direction bits

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

Reset:
  DEBUG CLS

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

Main:
  SEROUT Sio, Baud, ["!MC", Addr, "V"]
  SERIN  Sio, Baud, 100, Bad_ID, [STR id\3]
  DEBUG "MC2 Version ", STR id\3, CR, CR

  DEBUG "Ramping Up M1", CR
  FOR speed1 = 0 TO 255 STEP 5
    SEROUT Sio, Baud, ["!MC", Addr, "S", 1, speed1, MRev]
    PAUSE 200
  NEXT

```

## Column #144: Rev It Up

```
DEBUG "Hold @ 100%", CR
GOSUB Show_Status
PAUSE 2000

DEBUG "Ramping Down", CR
FOR speed1 = 255 TO 50 STEP 5
  SEROUT Sio, Baud, ["!MC", Addr, "S", 1, speed1, MRev]
  PAUSE 100
NEXT

DEBUG "Hold @ 20%", CR
GOSUB Show_Status
PAUSE 2000

DEBUG "Stop Motors", CR
SEROUT Sio, Baud, ["!MC", Addr, "X"]
GOSUB Show_Status

DEBUG "Done."
END

Bad_ID:
DEBUG CLS, "No response from address: ", IBIN2 Addr
END

' -----[ Subroutines ]-----
Show_Status:
SEROUT Sio, Baud, ["!MC", Addr, "G"]
SERIN Sio, Baud, [STR speed1\3]
DEBUG "M1: ", DEC3 speed1, " ",
      (status.BIT0 * 12 + "F"), CR,
      "M2: ", DEC3 speed2, " ",
      (status.BIT1 * 12 + "F"), CR
RETURN
```