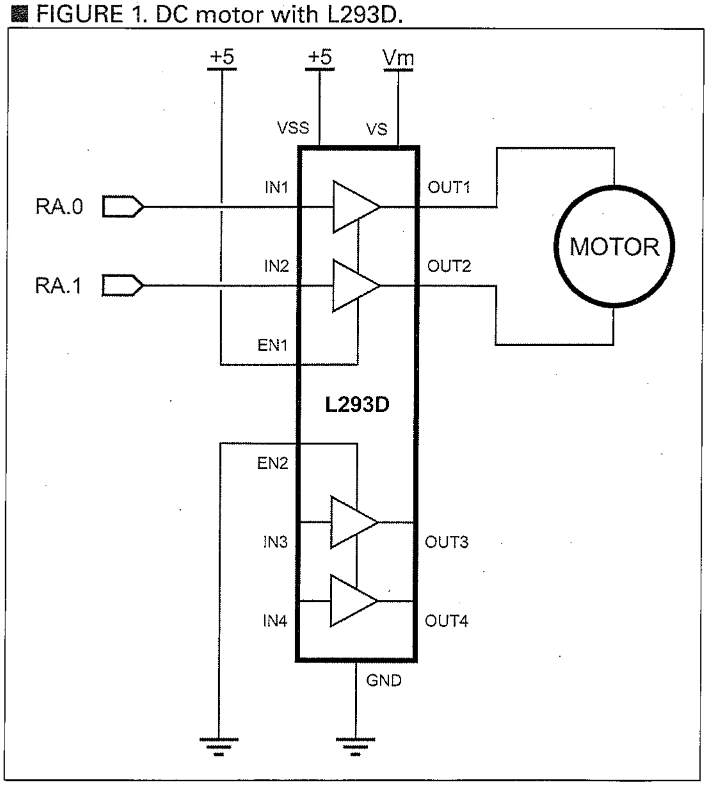
■ BY JON WILLIAMS

SX/B TURNS SWEET 16

THERE ARE THOSE — THE PESSIMISTS AMONG US — that will insist that you can't get anything worthwhile for nothing; everything has a price. Not so with SX/B. While it may not compete with big, "professional" compilers, in the right hands (i.e., yours) and with a few tricks, SX/B is quite capable and costs absolutely zero dollars. And with the cost of the SX-Key programming tool and SX Proto Boards so low these days, it's really hard to ignore the SX micro as a viable solution to many design problems.

Truth be told, it's easy for me to be a fan of SX/B because I was part of the team that developed it. Still, those of you who know me understand that I'm a very practical guy; I don't have a lot of time to fool around and when I need something, I need it, and I need it to work. Since leaving Parallax for new adventures, I have, in fact, continued to use SX/B — I recently designed a camera controller using an SX28 that was programmed entirely in SX/B 1.5x (no assembly language required). My point is that SX/B wasn't developed simply for the sake of doing it; SX/B was developed to provide a practical, no-cost tool for SX developers.

■ FIGURE 1. DC motor with L293D.



If you've never tried the SX, perhaps this article will encourage you to do so. It really is hard to beat the cost of entry: the SX-Key (ICP programming tool with full debugging capability) is only $79, the SX-Blitz (programming only, great for students) is an incredible bargain at only $29, and the fully-populated (power supply, SX chip, connectors) SX48 Proto Board is only $10! Yes, ten bucks. Using the Blitz, the SX48 Proto Board, a serial cable and a 12 VDC power supply, you could get into SX programming for about $50 — that's really not a bad deal for all the horsepower delivered by the SX.

SX/B 1.5X

The big news with version 1.5x is, of course, the addition of Word (16-bit) variables. This is especially good news for BS2 users wanting to port prototype projects to the SX for high-volume production. As in PBASIC, we declare a 16-bit variable in SX/B as type Word:

```
tmpW1           VAR      Word
```

When we look "under the hood" of SX/B (use Ctrl+L to compile and view the listing), we'll see that the definition above is actually composed of two bytes:

```
tmpW1           EQU      0x0D
tmpW1_LSB       EQU      tmpW1
tmpW1_MSB       EQU      tmpW1+1
```

Note that the value is stored "Little Endian" (low byte first) and that in addition to the name we declare, the compiler creates definitions with the suffixes _LSB and _MSB; these byte variables can be used just as we would use the *tmpW1.LOWBTYE* and *tmpW1.HIGHBYTE* notation in PBASIC.

Word variables can be used exactly as we'd expect — and even in a few ways that we might not consider at the start. The only caveat is this: Due to the limit of internal variables used by SX/B operations, we cannot multiply a word variable by itself and return the result to that same variable. The following line of code will generate a compiler error:

```
tmpW1 = tmpW1 * tmpW1
```

We can use the other operators here (+, -, /), just not operators that involve multiplication (*, */, and **).

Most of the SX/B instructions have been upgraded to work with Word variables, and a new variant of the **DATA** directive, called **WDATA** lets us store Word values for use with **READ**. The use of 16-bit values extends to I/O ports, as well. In SX/B 1.5x, there are three 16-bit pseudo ports: RBC, RCD, and RDE; the last two only apply to the SX45/52.

Here's a simple demo that shows how we can use the RBC port on an SX28:

```
Start:
  TRIS_B = %00000000
  TRIS_C = %00000000

  RBC = %00000000_00000001

Main:
  DO
    DELAY 75
    RBC = RBC << 1
  LOOP UNTIL RBC = %10000000_00000000
  DO
    DELAY 75
    RBC = RBC >> 1
  LOOP UNTIL RBC = %00000000_00000001
  GOTO Main
```

The program starts by making the RB and RC ports outputs — we have to do it this way because there is no TRIS_RBC port. The RBC port gets initialized and falls into a loop that simply ping-pongs the lit LED back and forth. Note the use of the underscore character in the comparison statement to make visualization of the 16-bit value easier.

Since this program uses a subroutine called ***DELAY***, and we might want to do delays with 16-bit values, let's look at the construction of subroutines in SX/B 1.5x.

For ***DELAY***, we'll use the following declaration:

```
DELAY           SUB     1, 2
```

This will let us pass a one- or two-byte value to ***DELAY***. Here's the actual subroutine code:

```
' Use: DELAY ms
' - 'ms' is delay in milliseconds, 0 - 65535

DELAY:
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1
  ELSE
    tmpW1 = __WPARAM12
  ENDIF
  PAUSE tmpW1
  RETURN
```

The construct of this subroutine is useful in many other situations as it allows us to pass bytes or words to the same subroutine. When we pass a byte, *__PARAMCNT* (internal variable) will be set to one by the compiler and the parameter is passed in *__PARAM1*. When we pass a word, *__PARAMCNT* will be set to two and the value passed in ***__WPARAM12***. Of course, we'll use a word-sized variable in the subroutine so that we can handle anything passed to it.

FUNCTIONAL SUBROUTINES

With the addition of 16-bit variables, a mechanism needs to be developed that would enable Word values to be returned from a subroutine; this is accomplished with the **FUNC** definition. This lets us define a function that can return up to four bytes (two words).

FUNC differs from **SUB** in that we will first specify how many bytes are to be returned, then the minimum and maximum parameter count used by the subroutine code for the function. Let's say that we wanted a function that would return a 32-bit product from two numbers — can we do it? Yes, of course! We don't have 32-bit values in SX/B, so we have to handle the words separately. Let's start with the function definition:

```
MULT32          FUNC    4, 2, 4
```

This definition says that the function, ***MULT32***, will return four bytes, and that the caller must pass between two and four bytes to it. This means that we could return the product of two bytes, a word and a byte, or two words. Note that the second option — multiply a word and a byte — can create some trickery for our subroutine construction, so we must make a decision about the order that values are passed. Let's decide that we will pass the word value first, then the byte. Here's the code for that function:

```
MULT32:
  IF __PARAMCNT = 2 THEN
    tmpW1 = __PARAM1
    tmpW2 = __PARAM2
  ENDIF
  IF __PARAMCNT = 3 THEN
    tmpW1 = __WPARAM12
    tmpW2 = __PARAM3
  ENDIF
  IF __PARAMCNT = 4 THEN
    tmpW1 = __WPARAM12
    tmpW2 = __WPARAM34
  ENDIF
  tmpW3 = tmpW1 ** tmpW2
  tmpW2 = tmpW1 * tmpW2
  RETURN tmpW2, tmpW3
```

As with the ***DELAY*** routine we did earlier, this code uses the *__PARAMCNT* variable to determine what is being passed and how to collect the parameters from the caller. The second choice, when *__PARAMCNT* is three, assumed that the first value passed is the word and the second is the byte. With the parameters collected, the rest is easy; the ** operator (new in SX/B 1.5x, and */ has been added, as well) returns the upper 16 bits from a 16-bit x 16-bit multiplication. The * operator will return the lower 16 bits of the product.

Note how the 32-bit value is returned to the caller as two words, separated by a comma, low-word first. So how do we collect this 32-bit value? Let's start with variables to hold it:

```
result          VAR     Word
resultHi        VAR     Word
```

And here's how we can use the function in a program.

```
  result = MULT32 $FFFF, $0100
  resultHi = __PARAM3, __PARAM4
  BREAK
```

The first part is obvious, I'm sure; we call the function and assign it to result. But this only gets us the lower 16 bits. To get the upper 16 bits, we have to grab them ourselves. The high word from the function will be returned in *__PARAM3* (LSB) and *__PARAM4* (MSB). This also demonstrates how to move two bytes into a word with just one line of code.

There is a method for collecting all four bytes from this function without the second line of code above — but we must use an array as the target variable. So, we could do this:

```
bigVal          VAR     Byte(4)
result          VAR     bigVal(0)
resultHi        VAR     bigVal(2)
```

And now we just need one line of code:

```
  bigVal = MULT32 $1234, $10
```

One of the interesting things about the SX-Key tool is that it will let us view 32-bit values in the Debug window. We can set up a **WATCH** declaration like this:

```
WATCH result, 32, UHEX
```

If we run the program in Debug mode with a **BREAK** instruction after the function call, we'll see the 32-bit result.

PIN DOWN YOUR I/O

One of the latest updates to SX/B is the **PIN** definition that became available in version 1.51. In the past, we might define an I/O pin like this:

```
Led             VAR     RC.0
```

Now we can do this:

```
Led             PIN     RC.0   OUTPUT
```

What's the advantage? Well, the compiler will automatically insert startup code that makes the pin an output, so we don't have to worry about anything beyond the declaration. That way, we can write directly to the pin knowing that the appropriate TRIS register has been set up correctly.

In a lot of my older programs, I would enable the SX pull-ups on any unused pin to minimize current draw. It's even easier now. Let's say that we have just the one LED as above. By using the following declarations, we don't have to worry about TRIS or PLP register settings in our code, which lets us focus solely on the application. Note how **PIN** works with groups and individual I/O pins.

```
UnusedA         PIN     RA      INPUT PULLUP
UnusedB         PIN     RB      INPUT PULLUP
UnusedC         PIN     RC      INPUT PULLUP

Led             PIN     RC.0    OUTPUT NOPULLUP
```

The final declaration overrides the definition for RC.0 from the group above; this way, we can define the unused pins as a group instead of one at a time.

It's important to understand that generation of **PIN** start-up code is enabled even when the **NOSTARTUP** option for the **PROGRAM** directive is specified. The available options for **PIN** are INPUT, OUTPUT, PULLUP, NOPULLUP, TTL, CMOS, and SCHMITT — and when multiple options are used, they are space-delimited.

INTERRUPTS WITHOUT IRRITATION

Before I get too far into this section, let me start by saying that interrupts are always tricky but that SX/B 1.5x does make them a bit easier to cope with. With SX/B 1.5x, we can simply specify how frequently (in interrupts per second) that the ISR should run and the compiler will take care of the rest, setting the OPTION register and the **RETURNINT** value automatically. Let's start with a very simple example:

```
' -------------------------------------------
  INTERRUPT NOPRESERVE 1000
' -------------------------------------------

ISR_Start:
  INC timer
  IF timer = Cycles THEN
    TOGGLE Led
    timer = 0
  ENDIF
  RETURNINT
```

The purpose of this code is to toggle the state of an LED every *N* milliseconds, defined by the program constant called **Cycles**. Note that the end of the **INTERRUPT** declaration line specifies 1000 — this will cause the program to set up the interrupt such that it runs once every millisecond. If we specify an ISR rate that that won't work with the **FREQ** directive, the compiler will complain of an invalid parameter.

This is interesting, but we may not want to blink the LED with a 50% duty cycle. Here's an easy update that allows us to specify the on- and off-time for the LED.

```
' -------------------------
INTERRUPT NOPRESERVE 1000
' -------------------------

ISR_Start:
  INC timer
  IF Led = IsOn THEN
    IF timer = OnTime THEN
      Led = IsOff
      timer = 0
    ENDIF
  ELSE
    IF timer = OffTime THEN
      Led = IsOn
      timer = 0
    ENDIF
  ENDIF
  RETURNINT
```

You've probably noticed that these programs use the **NOPRESERVE** option in the **INTERRUPT** declaration and may be wondering why and when to use this option. The reason why is that it will reduce the amount of code in the ISR. When can we use this option? We can use **NOPRESERVE** when none of the SX/B internal variables are being used in the ISR. This can be determined by using Ctrl+L to compile the program and show the assembly listing; if none of the internal variables (*__PARAM1 – __PARAM4*) are being used, then **NOPRESERVE** can and should be used.

Before we wrap up this section, let's take the second version of the LED blinker and use it to drive a motor. Remember the L293D that we used in the stepper project last month? Well, it's a push-pull driver so we can use two of its channels to drive a small DC motor and have control over speed and direction with just two I/O pins.

One pin will be pulse width modulated by the ISR to provide speed control. The other pin will determine the direction that the motor spins. We could add control of the L293D enable pin, as well, but this program assumes that it is tied high.

Let's look at the ISR first:

```
' -------------------------
INTERRUPT NOPRESERVE 10_000
' -------------------------

ISR_Start:
  INC phase
  IF phase > 100 THEN
    phase = 0
    IF m1Speed > 0 THEN
      M1Ctrl = IsOn
    ENDIF
  ELSE
    IF phase > m1Speed THEN
      M1Ctrl = IsOff
    ENDIF
  ENDIF
  RETURNINT
```

Looks pretty simple, doesn't it? In fact, it is. The code starts by incrementing a variable called ***phase*** — this tracks where we are — 0 to 100% — in the PWM cycle for the motor. When that value exceeds 100, we reset everything by clearing the phase counter and turning the motor control output on (if the speed is not set to zero). During the rest of the cycle, we compare the phase value to the speed of the motor; as soon as ***phase*** exceeds the motor speed, the motor is shut off. The behavior of this code lets us specify the motor speed in percentage.

The ISR runs the motor, but we need a subroutine to set the speed and direction when we need a change.

```
SET_MOTOR:
  tmpB1 = __PARAM1
  tmpB2 = __PARAM2
  m1Speed = tmpB1 MAX 100
  IF tmpB2 = Fwd THEN
    M1Dir = Fwd
  ELSE
    m1Speed = 100 - m1Speed
    M1Dir = Rev
  ENDIF
  RETURN
```

This code, too, is very straightforward. After collecting the parameters, the speed is set, limiting its value to 100. Then the direction pin (second motor control output) is set. Here's where we need to make an adjustment when reverse direction is specified. The ISR always makes the motor control pin high during the "on" phase of the motor. This is fine when the direction is set to forward and the direction pin is low, but when the direction pin is high (for reverse), what was the "on" time of the motor actually becomes the "off" time. Don't worry, the solution is simple. All we have to do is "invert" the reverse speed value by subtracting it from 100.

From my point-of-view, motor PWM control is a bit of black art. Luckily, the code is really easy to update. I found that setting my ISR rate to 10,000 (which works out to a 100 Hz PWM frequency) worked best for the motor I was using. If this setting was too high, the motor wouldn't move at low speeds; if it was too low, the movement was very choppy at low speeds. You may need to experiment with your motor.

Finally, we must remember that when the ISR is enabled as in the previous examples, it "steals" time from the rest of our program and will affect time-sensitive instructions like **PAUSE** and **PAUSEUS** (they get longer), and **SERIN** and **SEROUT** may not work at all. Advanced programmers will appreciate the *Effective-Hertz* parameter of the **FREQ** directive in SX/B 1.5x. If the ISR code runs a fixed period, then we can determine the "effective" clock frequency when the ISR is active and allow the compiler to generate code that will operate as expected.

SX/B WITH STYLE

In the SX/B 1.5x help file, you'll find a section called "The Elements of SX/B Style." This was, of course, adapted from "The Elements of PBASIC Style" that appears on the Parallax website and in the PBASIC help file.

The key to success with SX/B, I believe, is using subroutines and functions properly. If you do this, for example:

```
SERIN char1
SERIN char2
SERIN char3
```

You'll chew up a whole bunch of code space as each **SERIN** instruction is expanded to the assembly code required for that function — there is no automatic optimization by the compiler. Optimization, then, is the responsibility of the programmer, and the easiest way to do it is put "big" instructions into subroutines and functions.

What's a "big" instruction? It is any instruction that expands to more than a few lines of assembly code; most of the instructions that have any sort of timing element will fall into this category, things like **SEROUT**, **SERIN**, **PAUSE**, etc.

One final note on **SUB** and **FUNC** declarations: When the subroutine code does not require any parameters, use 0 in the declaration — like this:

```
RX_BYTE     FUNC     1, 0
```

This will save a bit of generated code — just a bit — but every little bit counts with small micros, right?

SX/B 1.5x has a couple more tricks up its sleeve you'll like: **COUNT** and **COMPARE** instructions (ala BS2), **TIMER1/TIMER2** instructions that simplify the use of the SX48/52 multi-purpose timers, and an option I particularly like is the clock speed multiplier for **SHIFTIN** and **SHIFTOUT**; this lets us connect to synchronous devices at (or very near) their maximum clock speed.

It's your turn now; if you're already using SX/B, make sure you download the latest version (it's free!), and if you're not using the SX, why not? When one considers the cost of entry, a free compiler like SX/B, and the horsepower the chip can deliver ... in my book, it's a great value and should be part of your arsenal. Give it a try — you'll be glad you did.

So good-bye for now and, until next time, Happy Stamping — SX/B style! **NV**

JON WILLIAMS
jwilliams@efx-tek.com
PARALLAX, INC.
www.parallax.com