Column #121 May 2005 by Jon Williams:

The Parallax Professional Development Board

When I first started working for Parallax my boss, Ken, handed me a big development board and said, "See what you can do with this." The board was, of course, the original NX-1000 and Ken and I went on to create StampWorks around it. Even after StampWorks, the NX-1000 never left my desk – it was the perfect "playground" for my experiments and many of my projects for this column originated on the NX-1000 development board.

The NX-1000 went through a series of improvements and ultimately ended up as two versions: a standard model and one that supported 40-pin BASIC Stamp modules. Not too long ago my colleague, John Barrowman, was working with an NX-1000 and felt like there was an opportunity to integrate all the features we liked in the existing models, plus a few new tricks. John was right, and after a lot of hard work and feedback from me and the other "power users" in the office, the Professional Development Board (PDB) was born. Figure 121.1 shows the PDB in all its glory.

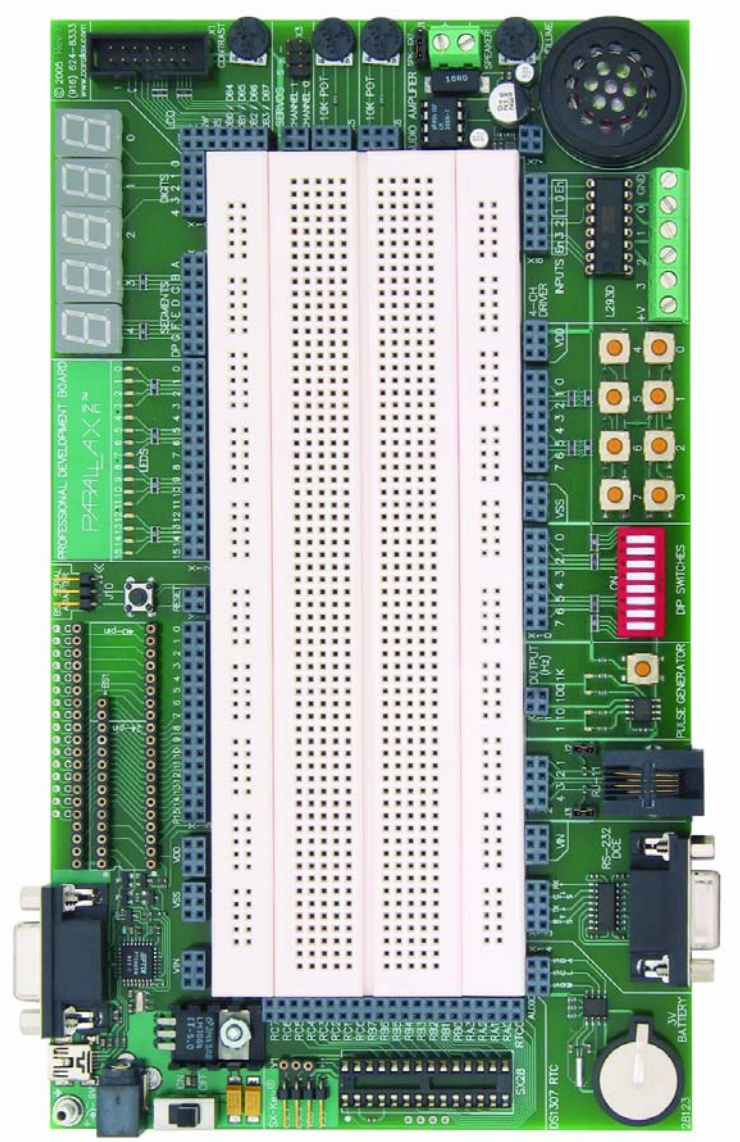

Figure 121.1: The new Parallax Professional Development Board

In a word, the PDB is simply stellar; it has become my favorite development platform – you'll see why as I work through its features. And don't get caught out by the term "Professional" in its moniker; that's referring to its professional features, not the price tag. As a matter of fact, the price of the PDB is actually lower than the NX-1000 – we can thank the engineering and manufacturing folks at Parallax for delivering such a great platform at such a reasonable price (about $150).

And just a note to my friends who are using "the other guys" 24-pin microcontrollers (products like the OOPic and BX-24): I think the PDB will work fine for you too – but please, try before you buy because Parallax hasn't tested the programming connections with anything but their own products.

Modules, Modules, Modules

What really excites me about the PDB is that I can develop projects for any of the microcontroller modules that I would normally work with: the BS1 (SIP version), the BS2 family (24- and 40-pin versions), the Javelin Stamp, and even the SX28. Even better, when using the BS1 or a 24-pin BASIC Stamp module, the SX28 can be used at the same time. Remember the LED controller I did back in January using the SX28? I built and tested that project with a BASIC Stamp master on a very early prototype of the PDB. The only caution one must take regarding the SX28 is when you want to work with the BS2p40. The AUX lines of the BS2p40 share the same terminals as the RB and RC ports of the SX28. So, if you want to use the BS2p40, be sure you remove the SX28 first so that there is no conflict with the IO pins. Figure 121.2 shows how you can mix-and-match the various microcontroller modules.

	BS1-IC	BS2-24	BS2-40	Javelin	SX28
BS1-IC		×	×	×	✓
BS2-24	×		×	×	✓
BS2-40	×	×		×	×
Javelin	×	×	×		✓
SX28	✓	✓	×	✓	

Figure 121.2: Parallax Microcontroller Compatibility on the PDB

The next logical question is programming ports; the PDB has four – yes, four: a USB serial port, a DB-9 serial port, a 3-pin header for the BS1 Serial Adapter, and a 4-pin header for the SX-Key. Figure 121.3 shows the programming connections and how they work with the various modules. Suffice to say, the PDB has all the programming connections we'll need to get the task done; just note that the USB and DB-9 programming connections cannot be used at the same time.

	USB	DB-9	BS1-SA	SX-Key
BS1-IC	✓		✓	
BS2 Family	✓	✓		
Javelin Stamp	✓	✓		
SX28AC/DP				✓

Figure 121.3: Programming Connection Options

Parts Galore

Like its cousin the NX-1000, the PDB has a whole host of support components that make project development a breeze. And when we need something that is not actually part of the PDB, there's a full-sized solderless breadboard that gives us ample space for extra components.

Here's the complete list of the on-board connections and support components, as they appear on the PDB moving clockwise from the upper left-hand corner:

- USB port (programming)
- DB-9F (programming)
- 24/40-pin socket
- 14-pin SIP socket (BS1-IC)
- 3-pin BS1 port (programming)
- Reset button
- (16) discrete LEDs
- (5) 7-segment LEDs
- Parallel LCD connection

- (2) 3-pin servo headers
- (2) 10K pots
- LM386 audio amp
- L293D push-pull driver
- (8) active-low push-buttons
- (8) active-low DIP switches
- Pulse generator
- RJ-11 connector
- DB-9F serial connector
- DS1307 RTC
- SX28AC/DP socket
- 3-pin socket for SX resonator
- 4-pin SX-Key header
- Power switch
- 2.1 mm power input

Yep, that's a whole lotta good stuff – and remember that this surrounds a large breadboard. Since most of those components are commonplace and we use them day-in and day-out, I don't need to get into any real detail, but let's do cover the essentials:

LEDs: The 16 discrete LEDs and five 7-segment LED modules are blue. There's nothing magic, they just look doggone cool. All LEDs are active high, so for the 7-segment displays you will enable a digit by pulling its common-cathode line low.

LCD Port: This is a standard 2x7 box header that we can use with common LCDs. One thing of note is that the socket for the signal lines splits the LCD bus – the DB0-DB3 pins (not used in 4-bit mode) are on the left, and the DB4-DB7 lines are on the right. If you look very closely at Figure 121.1 you can see the markings adjacent to the connector. Keep this in mind when you're making LCD connections. You can use either 4-bit or 8-bit mode, you just have to use the appropriate connections in the socket.

Servo Ports: There are two 3-pin headers that can be used with servos or devices that follow the same connection layout (Signal-Vdd-Vss). For example, the Parallax Serial LCD module or PING))) sonar sensor could be connected to these ports using a standard servo extender cable.

Pots: There are two 10K pots with all connections made available. And if you look very closely you'll also see a small resistor (220 ohm) inline with the wiper connection. This is one of the nice features John built into the PDB. This resistor prevents the accidental connection of an IO pin directly to Vss or Vdd; a connection that when coupled with a

programming error could damage the BASIC Stamp or SX. This feature makes the PDB "expert proof." Come on, admit it, we've all made wiring errors and damaged components – that's less likely with the PDB.

Audio Amp: This is a fairly standard LM386 audio amp circuit with a built-in filter that makes it suitable for the SOUND (BS1), FREQOUT and DTMFOUT instructions. A small speaker is built right into the PDB, but if you want more volume you can move the speaker jumper to EXT and connect a small 8-ohm speaker. I use a nice little bookshelf speaker from RadioShack® and it sounds great.

L293D: The NX-1000 used the ULN2803 for high current outputs; the PDB has changed to the L293D. The nice thing about the L293D is that it's a push-pull driver (sources and sinks current on outputs); the ULN2803 only sinks current. The ability of the L293 to source and sink current gives us more flexibility – channels can be combined to form an H-Bridge circuit. Since the L293D is new (to me), we'll do a simple project with it after we get through the parts description.

Push buttons: The PDB has eight, active-low (each IO pin gets pulled to Vdd through 10K) push buttons. As with the 10K pots, the push button circuits have inline 220-ohm resistors to protect IO pins.

DIP Switches: Eight, active-low DIP switches – electrically identical to the push buttons.

Pulse Generator: The PDB pulse generator is identical to that on the NX-1000; it provides a 50% duty cycle pulse at 1 Hz, 10 Hz, 100 Hz, or 1 kHz. A push button is used to set the output frequency.

RJ-11: This connection has configuration jumpers so that it can be used with the iButton® "Blue Dot Receptor" modules (for OWIN and OWOUT), as well as X-10 transmitter modules (for XOUT).

DB-9F Serial Port: As with the early version of the NX-1000, this lets us connect to off-board serial devices. The level-shifter gives us TX, RX, RTS, and CTS lines, which means we can implement full flow control (important for many projects).

DS1307 RTC: This is the same I2C™ RTC that is used on the NX-1000-24/40. A lithium backup cell is included.

All in all, this is a very nice complement of components and flexibility, and does indeed give us the freedom to create to our hearts' desire. As much as I like my NX-1000 boards, the

PDB has replaced them as my favorite development tool – I think when you give it a try it will be yours too. Okay, enough chit-chat, right? – let's do something.

Steppin' Out With the L293D

Stepper motors are fairly easy to control with the BASIC Stamp, and with the L293D we can now control either unipolar (5- or 6-wire) or bipolar (4-wire) stepper motors. This is very cool; I frequently find low-cost bipolar steppers at Tanners – now I can put them to use. And here's even better news: the same program will control either motor type!

That last part really made me smile. I had never used a bipolar stepper before so I did a little Googling to see what I was getting into. Imagine my surprise when I found the step table for bipolar steppers exactly matched what I was already doing with unipolar motors. Now, I'm a "Show me" kind of guy, so I tried … it worked! … way cool!

For fun I'm going to present the program for the BS1 – though it is super easy to modify for the BS2. Why the BS1? Well, I've been spending a lot of time with the BS1 lately. In February John and I went to a Halloween-oriented convention called Death Fest (www.deathfest.net). It turns out that there are lots of interesting folks that build Halloween (and other holiday) displays, and many use the BS1 to control things. In fact, there were so many and their needs were so specific that John and I created a repackaged version of the BS1 called the Prop-1 Controller – it's a BS1 core with an onboard ULN2803 and a few other features that help prop and FX builders.

So, contrary to the belief by some, the BS1 is not dead – in fact the 2.1 version of the BASIC Stamp Editor handles the BS1 very nicely, adding the Memory Map feature that used to be restricted to the BS2. The Memory Map is really important for the BS1 as its EEPROM space is so small. The other thing that I like is it will remind me that I can't use W6 for variable space when I declare subroutines (because W6 gets used as the GOSUB-RETURN stack).

Okay, onto the stepper code. Since the program is very short, and we haven't used the BS1 in a long time, I'm going to go through every part of the program in some detail – this will get you back into the saddle with the BSI or, perhaps, get you interested if you haven't used it in the past.

Let's look at the declarations first. In the BS1, everything is declared as a SYMBOL; IO pins, constants, and variables.

```
SYMBOL  Stepper         = PINS
SYMBOL  ChngDir         = PIN7

SYMBOL  StpsPerRev      = 48
SYMBOL  NotPressed      = 1
SYMBOL  Pressed         = 0
SYMBOL  CW              = 0
SYMBOL  CCW             = 1

SYMBOL  rotation        = BIT0
SYMBOL  idx             = B2
SYMBOL  phase           = B3
SYMBOL  stpIdx          = B4
SYMBOL  stpDelay        = B5
```

The BS1 does not have separate names for the output and input registers (OUTS and INS in the BS2) – there is a register called PINS that is contextually sensitive. When we write a value to PINS, that value goes to the output register; when we read a value from PINS, we're getting that value from the input register.

Notice that the declaration for Stepper uses the entire PINS register – we need to do this as there is no way to split PINS into nibble-sized groups. Since our program actually needs just four pins for the stepper outputs, we'll use a mask to protect the pins not connected to the motor. More on that later.

The PINS register can, however be declared as individual bits. For this program we're going to use a push-button to set the rotational direction of the stepper; we'll connect this button to P7, hence the PIN7 declaration for ChngDir. Note that some commands in PBASIC 1.0 require the pin number as a constant value, so keep that in mind when declaring symbols for IO pins, and be sure to double-check the help file as to whether a command wants an IO register variable (e.g., PIN5) or the pin port number (e.g., 5).

As you can see, constants are very easy to define – we simply put the symbol's value after the equal sign. The BS1 does allow 16-bit values (like the BS2), so a constant can have value from zero to 65535.

Finally, we should declare variables with meaningful names – and this can be the trickiest thing to do with the BS1. In the BS1 we have to manually assign our variable names to a specific location in RAM. For most programs we will be using byte variables, so I recommend that you start your definitions at B2. Why? Well, the BS1 only lets us deal with 16 bit-sized variables, and those are attached to W0 (B0 and B1). If we start our byte definitions at B2 then the bit variables will be available if/when we need them. For example,

the original version of the stepper program didn't have the reverse button. Adding the button and the rotation variable to keep track of direction was no trouble because neither B0 nor B1 had been previously assigned.

I know what you're thinking … "If I have to map variables manually why don't I just use the built-in RAM names?" It's tempting, I know, but I can't even begin to remember how many customer programs I've fixed that had crossed-up variables because the internal names were used. Don't do it ... don't go to the dark side … it just takes a few minutes to define useful names, so trust me and just do it.

For review, we can have BIT0 to BIT15, B0 to B12, and W0 to W6. Not a lot of variable space, I know, but then we don't have a lot of code space either so it's usually not a problem. Just remember that BIT0 to BIT7 occupy the same space as B0, which is actually the low byte of W0 – now you can see why the Memory Map is so important for the BS1. Figure 121.4 shows the Memory Map for the stepper program. Note how W6 is colored yellow to indicate its use for GOSUB-RETURN. If we attempt to assign a variable to W6 (or B11 or B12), the Memory Map will show yellow with a red slash to indicate a conflict. This issue only comes up when using GOSUB-RETURN – if that doesn't exist in a program then we are free to use the W6 RAM space.

I tend to separate my setup from main loop code to keep things organized. In this program there are just two things to do: initialize the motor pins to be outputs and set the motor speed.

```
Reset:
  DIRS = %00001111
  stpDelay = 3
```

No magic here, the BS1 has a DIRS register just like the BS2 – the difference being that the BS1 register is only eight bits wide. The variable called stpDelay (delay between steps) affects how fast the motor turns; the smaller the value, the faster it turns.

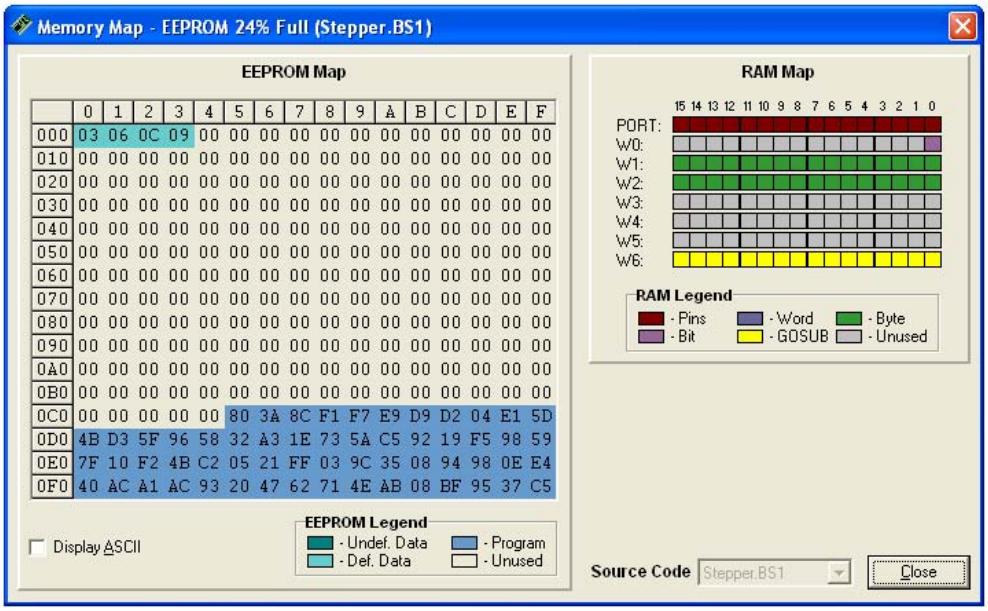

Figure 121.4: Memory Map for Stepper Program

And now we get to the heart of the program. The logic is straightforward: check the change-of-direction input; if it's pressed we'll invert the rotation flag and wait for the button to released, and finally jump to the code that rotates the motor in the current direction. If the change-of-direction button isn't pressed we jump right to rotating the motor.

```
Main:
  IF ChngDir = NotPressed THEN Move_Motor
  rotation = rotation ^ 1

Force_Release:
  IF ChngDir = Pressed THEN Force_Release

Move_Motor:
  IF rotation = CCW THEN Turn_CCW

Turn_CW:
  GOSUB Step_Fwd
  GOTO Main

Turn_CCW:
  GOSUB Step_Rev
  GOTO Main
```

If you're not accustomed to BS1 code the first thing that stands out is the number of program labels. The BS1 doesn't have the IF-THEN-ELSE construct, so we have to apply a bit of inverted logic to get IF-THEN to flow smoothly. It's not difficult, it just takes a little planning (very little!). I think that one read-through of the code will make perfect sense and, if you take away nothing else, please note the use of appropriately named constants and variables makes the program very easy to follow. This is why I get so adamant about defining meaningful symbols – it makes the rest of our work easier.

The final section holds the subroutines to actually move the motor. This code is pulled out as subroutines (instead of being buried in the main program loop) so that it can be used in other programs.

```
Step_Fwd:
  stpIdx = stpIdx + 1 // 4
  GOTO Do_Step

Step_Rev:
  stpIdx = stpIdx + 3 // 4
  GOTO Do_Step

Do_Step:
  READ stpIdx, phase
  Stepper = Stepper & %11110000 | phase
  PAUSE stpDelay
  RETURN
```

There are two separate subroutines for movement, but their purpose is the same: calculate the next step in the sequence so that we can pull the coil data from a table. The key to both of these routines is the modulus operator (//). I love modulus; my boss on the other hand, he hates it. Okay, hate is a strong word – let's just say the modulus operator is not his favorite. Don't worry, I'm still working on him.

Modulus is really quite simple: it returns the remainder of a division. In practice this means that the modulus operator will return a value between zero and the divisor minus one. In our program, for example, we're going to control the stepper with four whole steps so we'll want to keep the table index between zero and three. The modulus operator is perfect for this. To point to the next position in the table we'll add one to the index and then take the modulus of four. The fact that modulus will cause a wrap-around to zero when we get to the end of the table is particularly useful. Without modulus, we'd have to do this:

```
stpIdx = stpIdx + 1
IF stpIdx < 4 THEN Do_Step
stpIdx = 0
GOTO Do_Step
```

Do you see how much easier using modulus is? Going backward is the same but requires just a little bit of thinking. Instead of adding one, we're going to add the value of the divisor minus one (three in our program as the divisor is four). Let's look at real values to be clear:

0 + 3 = 3 ... 3 // 4 = 3
3 + 3 = 6 ... 6 // 4 = 2
2 + 3 = 5 ... 5 // 4 = 1
1 + 3 = 4 ... 4 // 4 = 0

With the new index in place we will read the coil data from a table. Here's how that's defined:

```
Full_Steps:
  EEPROM 0, (%0011, %0110, %1100, %1001)
```

Note that we can place a table at the specific location (first value after EEPROM), but we can't use program labels to define table locations – we have to keep track of them manually. That's not a problem here as we only have one table.

Have a look the subroutine called Do_Step – this does the actual work. The first line uses READ to retrieve coil data from our EEPROM table. The next line writes this data to the stepper while preserving the condition of P4 – P7. This is done with a mask value (%11110000). Since the upper four bits of the mask are ones, and we're using the AND operator (&), the current values of those pins will be preserved. With the lower four bits of the mask being zero, the coil outputs will be cleared. The last part of that line adds the new coil data to the outputs.

Finally, the speed control variable, stpDelay, is used with PAUSE and then the process starts all over again.

Getting Connected to the PDB.

Now that we've worked through the code I should probably give you a couple hints about connecting your stepper to the PDB. The connections between the BASIC Stamp and the L293D inputs are obvious; P0 – P3 connects to INPUTS 0 – 3 on the PDB (IN1 – IN4 on the L293D). There are two enable pins – we need to pull both high to enable the outputs (each enable pin controls two outputs; when low, the respective outputs go to a Hi-Z state). Power for the motor is routed from the Vin pin (by the secondary RS-232 port) to the +V terminal near the L293D. Using an external +V connection lets us supply a voltage appropriate for the motor we'll connect.

The last step is to connect the stepper. With a unipolar motor we will connect the phase connections to the L293D outputs and the common wire to +V. With a bipolar motor we simply connect the coils.

My unipolar motor is made by Mitsumi and is marked M42SP-5 – you can get this motor from Parallax. The bipolar motor is made by Minebea and is marked 17BB-H132-11. I bought mine at Tanners in Dallas, but I've seen them online with surplus dealers. The schematic for the circuit is shown in Figure 121.5, and the color codes for the motors I tested are shown in Figure 121.6.

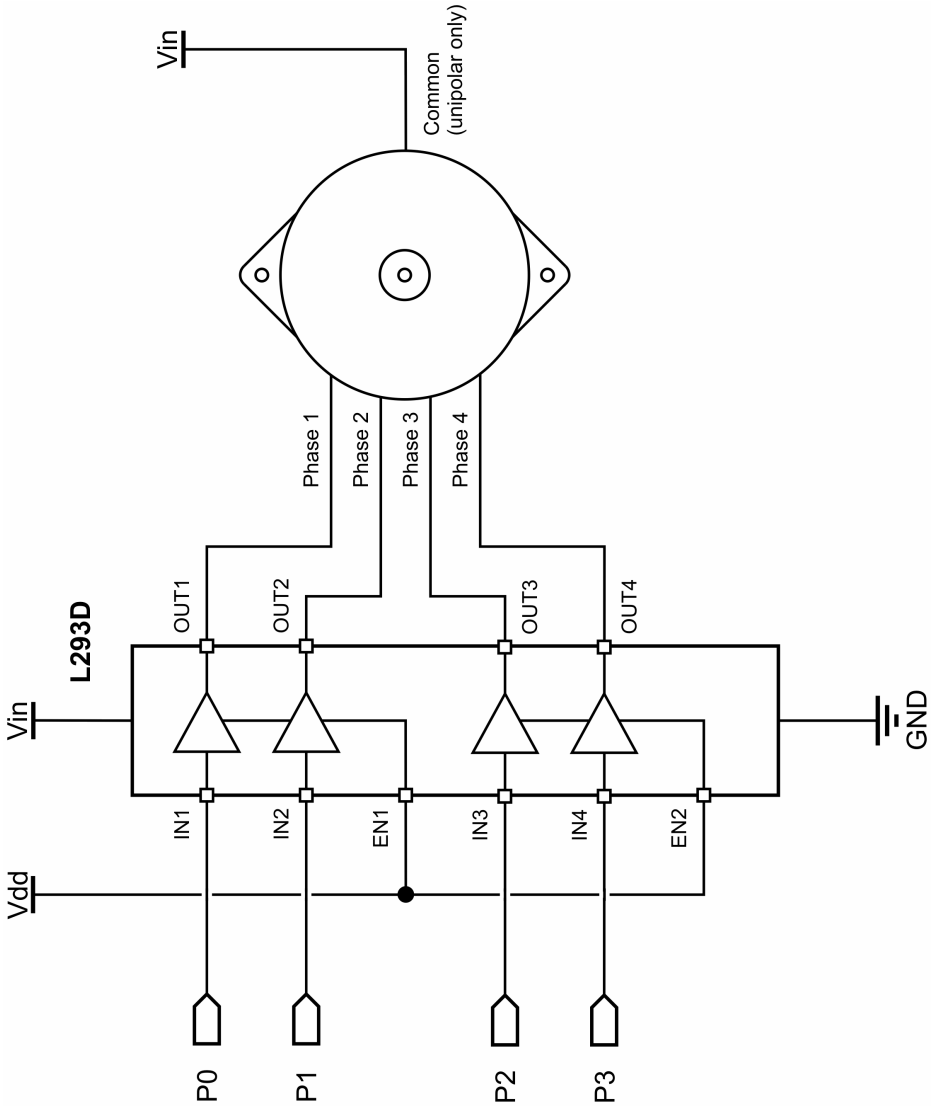

Figure 121.5: L293 Stepper Schematic

	+V	Ph 1	Ph 2	Ph 3	Ph 4
Unipolar	Red	Black	Orange	Brown	Yellow
Bipolar	N/C	Red	Orange	Yellow	Brown

Figure 121.6: L293 Stepper Color Code

Before I wrap it up, let me share a hint about bipolar steppers that I found during my research – you can follow these steps when you don't know which wire corresponds to which coil (phase):

- Connect the wires in any order
- Run the program, if it works, you're done
- If not, reverse the outer two leads (Ph1 and Ph4)
- Run the program, if it works, you're done
- If not, reverse the inner two leads (Ph2 and Ph3)
- The program will now work

I actually used these steps with my motor, and did in fact have to go to the very end. It did work, though, and save me a lot of trouble experimenting.

That's it for this month. Have fun with the PDB and the microcontroller of your choice and, until next time, Happy Stamping.

```
' ================================================================================
'
'   File...... Stepper.BS1
'   Purpose... Simple Stepper Motor Demo
'   Author.... Jon Williams, Parallax Inc.
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 18 MAR 2005
'
'   {$STAMP BS1}
'   {$PBASIC 1.0}
'
' ================================================================================


' -----[ Program Description ]----------------------------------------------
'
' Stepper motor demo using the BS1 and L293D push-pull driver.  The nature
' of the L293D allows this code to control both unipolar and bipolar
' stepper motors.
'
'
' Connections (BS1 --> L293D --> Mitsumi M43SP-5, unipolar, 5 wires)
'
' P0  --> IN1/OUT1 --> Black
' P1  --> IN2/OUT2 --> Orange
' P2  --> IN3/OUT3 --> Brown
' P3  --> IN4/OUT4 --> Yellow
' Vdd --> Enable1
' Vdd --> Enable2
' Vin --> +V       --> Red


' Connections (BS1 --> L293D --> Minebea 1788-H132-11, bipolar, 4 wires)
'
' P0  --> IN1/OUT1 --> Red
' P1  --> IN2/OUT2 --> Orange
' P2  --> IN3/OUT3 --> Yellow
' P3  --> IN4/OUT4 --> Brown
' Vdd --> Enable1
' Vdd --> Enable2
' Vin --> +V


' -----[ Revision History ]------------------------------------------------


' -----[ I/O Definitions ]--------------------------------------------------

SYMBOL  Stepper         = PINS                  ' uses P0 - P3
SYMBOL  ChngDir         = PIN7                  ' change direction pin
```

```
' -----[ Constants ]-------------------------------------------------------

SYMBOL  StpsPerRev      = 48                    ' whole steps per rev

SYMBOL  NotPressed      = 1
SYMBOL  Pressed         = 0

SYMBOL  CW              = 0
SYMBOL  CCW             = 1


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

SYMBOL  rotation        = BIT0
SYMBOL  idx             = B2                    ' loop counter
SYMBOL  phase           = B3                    ' new phase data
SYMBOL  stpIdx          = B4                    ' step pointer
SYMBOL  stpDelay        = B5                    ' delay for speed control


' -----[ EEPROM Data ]-----------------------------------------------------

Full_Steps:
  EEPROM 0, (%0011, %0110, %1100, %1001)        ' step data


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

Reset:
  DIRS = %00001111                              ' make P0..P3 outputs
  stpDelay = 3                                  ' set step delay


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

Main:
  IF ChngDir = NotPressed THEN Move_Motor       ' skip if button not pressed
  rotation = rotation ^ 1                       ' reverse motor direction

Force_Release:
  IF ChngDir = Pressed THEN Force_Release       ' hold while button pressed

Move_Motor:
  IF rotation = CCW THEN Turn_CCW               ' check current rotation

Turn_CW:                                        ' do one step CW
  GOSUB Step_Fwd
  GOTO Main
```

```
Turn_CCW:                                       ' do one step CCW
  GOSUB Step_Rev
  GOTO Main

  END


' -----[ Subroutines ]------------------------------------------------------

' Turn stepper clockwise one full step

Step_Fwd:
  stpIdx = stpIdx + 1 // 4                      ' point to next step
  GOTO Do_Step


' Turn stepper counter-clockwise one full step

Step_Rev:
  stpIdx = stpIdx + 3 // 4                      ' point to previous step
  GOTO Do_Step


' Read new step data and output to stepper pins while preserving
' the state of other IOs

Do_Step:
  READ stpIdx, phase                            ' read new phase data
  Stepper = Stepper & %11110000 | phase         ' update stepper pins
  PAUSE stpDelay                                ' pause between steps
  RETURN
```