Column #136, August 2006 by Jon Williams:

Stepping Out with Spin

Are you ready to get some motors spinning? Yeah, me too. As we saw last month with the BS1, to get more performance we have to think differently. With the Propeller chip, the entire paradigm is different and we are in fact forced to adapt. Sometimes change can be a bit uncomfortable, and changing from one programming style to another can be, well… tough -- if we let it. The funny thing for me is that I have done so much Spin programming the past year that when a client asked me to design a control product using an SX28, I had to shift gears back to a somewhat linear style. Yes, there were a few moments when I was wishing the customer had wanted to use the Propeller – my life might have been a little easier. Okay, then, how do we make the shift to Spin programming so that it becomes as comfortable as our old friend, PBASIC? For me, direct translation is the best way to start.

Perhaps it was me being too busy to notice, but I have this sudden awareness that controlling stepper motors is popular again. They are indeed very cool, finding use in hobby applications in everything from robotics to home-built machine shop tools. And stepper control is actually quite easy as we've seen in the past. Let me qualify that controlling one stepper is quite easy; what if we want to control two – independently – for a robot, or even three for some kind of XYZ machine platform? Yes, we could resort to a hardware stepper controller, but with the Propeller… we don't have to anymore; we can code our own controller in Spin and manage as many motors as our I/O pins will allow.

First things first: let's take the standard BS2 stepper demo program and translate it to Spin. This will do a couple things for us: 1) It lets us easily verify that the hardware is working with

a simple program, and 2) It helps us make the translation from something we know to something we want to know.

Translation is on my mind lately as on my morning walks I listen to a Japanese language training CD. With the CD I am taught new words and phrases, and then a bit later the "teacher" will ask, "How do you say…." We can do this for ourselves. I'll be your teacher for this lesson; after you've mastered it, I suggest you follow this same process with your favorite BASIC Stamp programs.

From PBASIC to Spin

To start with I'm going to translate the BS2 stepper motor demo code that is available on the Parallax web site (and in the ZIP for this article). We'll go section-by-section making the translation from PBASIC to Spin.

Figure 136.1 shows the driver that we'll use to run the stepper motor: an L293D. I like this chip for driving steppers because it can drive unipolar (5 or 6 wires) or bipolar (4 wires) steppers with the same code. The only difference in the connections is that the unipolar motor has a common connection (or two) that goes to GND. Note that the original BS2 demo does not deal with the L293D enable pin, so we're going to add that to the program – it's a useful feature as it allows the stepper to "coast" when enable is not active.

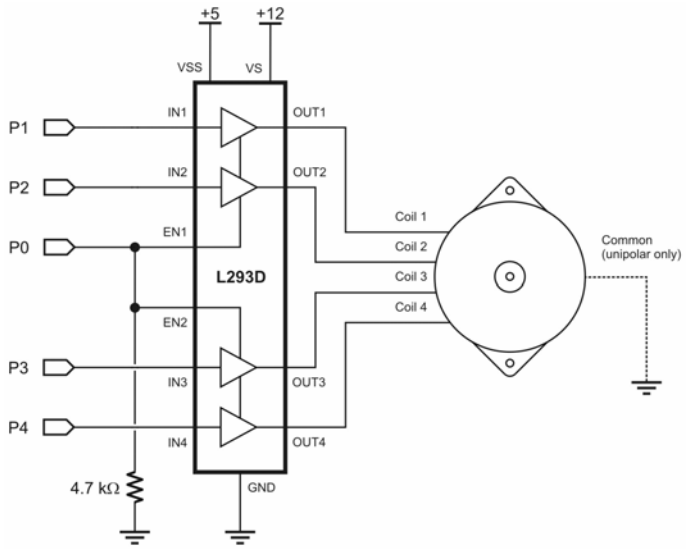

Figure 136.1: Stepper Motor with L293D

A couple things about the L293D: You'll not that the logic supply (+5) is called VSS by the manufacturer – do not connect this pin to ground. The GND pins (4, 5, 12, and 13 on the DIP version) are for ground. The motor supply pin is called VS, this is pin 12 on the DIP. Before you make the connections for the L293D version you select, be sure to review the data sheet so that you make the correct connections. Also note that I've pulled the L293D Enable pins low through a 4.7K resistor; this disables the outputs when the control pin is disconnected or floating. If the Enable pins are allowed the float the L293D outputs will be active.

Author's Note: The TI SN754410 can be used in place of the L293D.

Okay, let's start translating code. At the top of the BS2 program we'll find the following definitions for the I/O connections and the number of steps the motor requires for a single revolution.

```
Phase           VAR     OUTB

StpsPerRev      CON     48
```

And here's how we're going to end up translating that code to Spin:

```
CON

  _CLKMODE      = XTAL1 + PLL16X
  _XINFREQ      = 5_000_000

  EN            = 0
  M1            = EN + 1
  M4            = M1 + 3

  STEPS_PER_REV = 48
```

Okay, so there's a little more work involved with the Spin definitions, but with that we get a whole lot more power and flexibility. We always start by defining _CLKMODE and _XINFREQ; the listing shows our "standard" selection of a 5 MHz clock input and the 16x PLL tap; this causes the chip to run maximum speed (80 MHz).

Next come the I/O pins. As noted earlier, we're adding an Enable, along with the [contiguous] motor connections. With the Propeller we have enormous flexibility with I/O. On the BASIC Stamp, we select nibble boundaries for the four stepper motor outputs. With the Propeller, those boundaries don't exist and we can use any four contiguous pins to control the motor (we'll see that in just a second).

Next up are the global variable definitions:

```
idx             VAR     Byte
stpIdx          VAR     Nib
stpDelay        VAR     Byte
```

Remember that the BASIC Stamp uses an 8-bit core (PIC or SX), so its native variable type is the Byte. Bits, Nibs, and Words are allowed, but they actually take work on the inside of the Stamp to support them. The Propeller uses a 32-bit core, so its native variable type is the Long. Since the Propeller has 16x the RAM space of the BASIC Stamp, we don't have to be so conservative with variable definitions, in fact, by using Longs the program is more efficient as that is the native type.

```
VAR

  long  idx, stpIdx, stpDelay
```

Yes, we could have defined these variables as Bytes or Words (there are no Bit or Nib variables in Spin) but, again, it would not have improved the performance of the program.
The next section in our PBASIC version of the program is a DATA table for the stepper motor coil patterns.

```
Steps           DATA    %0011, %0110, %1100, %1001
```

And in Spin:

```
DAT

  Steps       byte      %0011, %0110, %1100, %1001
```

By convention, PBASIC DATA tables usually appear near the beginning of the program and Spin DAT section(s) appear at the end. This is just convention and the compilers do not care either way. Do note that we define the element size of items in a DAT table.

And now we get into the working code:

```
Setup:
  DIRB = %1111
  stpDelay = 15

Main:
  FOR idx = 1 TO StpsPerRev
```

```
    GOSUB Step_Fwd
  NEXT
  PAUSE 500
  FOR idx = 1 TO StpsPerRev
    GOSUB Step_Rev
  NEXT
  PAUSE 500
  GOTO Main
```

This is very simple; we start by making the motor control pins outputs, initializing the step timing (the delay, in milliseconds, between step changes), and then the stepper is rotated back and forth. The equivalent code in Spin – with a fun modification – looks like this:

```
PUB main

  dira[M4..EN]~~
  outa[EN]~~

  repeat
    stpDelay := 5
    repeat idx from 1 to STEPS_PER_REV
      stepFwd

    pause(500)

    repeat while (stpDelay < 50)
      repeat idx from 1 to STEPS_PER_REV
        stepRev
      stpDelay := stpDelay * 125 / 100

    pause(500)
```

As with the PBASIC program, our first task is to make the motor control pins outputs. And here's where Spin is very flexible with I/O pins. In the BASIC Stamp, we can work with one pin, four pins, eight pins, or all 16. With the Propeller, we can work with any contiguous group of pins that we choose by using the dot-dot notation. This line, then:

```
  dira[M4..EN]~~
```

...is equivalent to:

```
  dira[4] := 1
  dira[3] := 1
  dira[2] := 1
  dira[1] := 1
  dira[0] := 1
```

As you see, we've also used the post set to -1 (double tilde) operator as a shortcut. I find this best when the goal is to make all of the target bits one – it prevents possible errors by miscounting the number of bits in the target group. Do be sure to use two tildes as a single post tilde (~) will clear all target bits to zero.

After making the control pins outputs we set the enable pin high using the outa register. Again, the ~~ is used. You see, this operator works no matter how many bits are to be affected, hence it is incredibly useful.

And now we start running the motor. There is no GOTO in Spin, so the loop construct we use in the PBASIC version is handled with repeat. Look carefully, and you'll see that there is a single repeat under which everything else is indented. Remember, with Spin, formatting counts. If you're unsure about the levels of indenting in a program, the editor will show you by pressing [Ctrl]+[I] (this toggles the indention display on/off). And you can adjust the indent level of a group of lines by selecting them and pressing [Tab] (indent in) or [Shift]+[Tab] (indent out).

At the top of the loop we set the initial step delay which serves as a speed control for the stepper – the shorter the delay, the faster the stepper will move. Now don't get crazy and think that you can have nearly no step delay. Stepper motors are mechanical devices and do need a bit of delay betweens steps. If we start with a step delay that is too short the motor may not run, or will just sit there quivering; not a pretty sight….

Now we do the first [inner] loop that runs the motor forward one revolution. In PBASIC, FOR-NEXT is used; in Spin, we use repeat from. As you can see, there is no NEXT required because, again, it is indenting that defines the block of code for the loop. The only thing we're doing in the loop is moving the motor one step by calling a subroutine.

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

Step_Rev:
```

```
  stpIdx = stpIdx + 3 // 4
  GOTO Do_Step

Do_Step:
  READ (Steps + stpIdx), Phase
  PAUSE stpDelay
  RETURN
```

This, too, is simple code, and designed to absolutely be as small as possible. We don't have to work so hard in Spin, though we do have to create our own pause method to duplicate the PAUSE instruction of PBASIC.

```
PRI stepFwd

  stpIdx := ++stpIdx // 4
  outa[M4..M1] := Steps[stpIdx]
  pause(stpDelay)

PRI stepRev

  stpIdx := (stpIdx + 3) // 4
  outa[M4..M1] := Steps[stpIdx]
  pause(stpDelay)

PRI pause(ms) | c

  c := cnt
  repeat until (ms-- == 0)
    waitcnt(c += clkfreq / 1000)
```

This should all make perfect sense. Note the use of C-like operators: ++ (pre increment) and that our DAT table can be accessed like a simple array (hence no READ instruction).

The pause method here is quite useful and you'll probably want to have it for other programs. As you can see, this method uses local variables; these come from the stack and are not available to the program outside this method. For pause we will pass the number of milliseconds to delay, and we use waitcnt method to delay one millisecond within a loop.

This loop may look a bit tricky – unless you have a C or Java programming background – as a lot of work is being packed into one line of code.

```
  repeat until (ms-- == 0)
```

What is important to note here is the position of the decrement (--) operator; it comes after the variable so the comparison (ms == 0) will be done first. If the value of ms is greater than zero it will be decremented and the loop code will run. If we wanted to be very verbose we could expand the line to this:

```
repeat until (ms == 0)
  ms := ms - 1
```

Let's finish up and move on. If you look at the second half of the Spin version you'll see that we've simply added a speed change between each rotation. When the program runs the motor will spin one revolution forward, then several revolutions backward, slowing down as it does.

Mo' Motors, Please….

This is a lot of fun, but now it's time to move on and take advantage of the Propeller's eight cores – well, for the time being we'll just going to use one more. Our goal, of course, is to drop the stepper control into its own core so that it can run freely in the "background" while our "foreground" program is doing other things. By doing this our "foreground" is free, and with up to seven "background" cogs we can control a whole lotta motors!

If you're new to the Propeller please let me suggest that you go back and read the April, May, and June editions of this column, focusing especially on June where the idea of "background" processing and control is explored.

So we've decided to create a stepper motor object; we need to think about its behaviors. What features (behaviors) do we want in our stepper motor object? The basics, I think, should be something like this:

- Control the L293D enable pin
- Set motor to running or stopped
- Set mode to free running or step mode
- Set direction – forward or reverse
- Set motor speed through step delay
- Set discrete number of steps to run before stopping

That's enough to get us started with a fairly full-featured stepper controller. The object is called stepper.spin; go ahead and open it now so you can see how things are constructed.

The two key areas, of course, are the start method that is used to launch the control into its own cog, and the runStepper method that is in fact what does the actual work. To get runStepper into its own cog we will use cognew. If a cog was available and the method was

launched successfully, cognew will return a value greater than zero. Here's the start method that includes the launch code for runStepper:

```
PUB start(ePin, mnTime) : okay

  okay := cogon := (cog :=
    cognew(runStepper(ePin + 1), @stack)) > 0

  if okay
    en := ePin
    minStpTm := stepTm := mnTime
    setMode(%0000)
    enable(false)
    stpIdx~
    numSteps~
```

The start method expects two parameters: the pin that controls the L293D Enable inputs, and the minimum step timing for the motor being used. This method will return true or false depending on the result of cognew. As you'll see in the demo program, we won't actually do anything unless start returns a value of true.

When we launch runStepper with cognew we have to pass along the pin number of coil #1 for the motor – the others are expected to be contiguous. This is critical. I fought for nearly a day when writing the object because I couldn't get the background code to control the motor, while the foreground would happily do it.

Well, I finally gave in and made a call to Parallax. After going through the code with Chip and trying all sorts of things he suddenly blurted, "Oh, I see what's wrong!" Here it is… this is big… please remember it: Each cog has its own I/O pin definitions, and as runStepper runs in its own cog, it has to define any I/O pins that it needs to control. By passing the start of the motor pins group to the method it can set the pins as outputs and update the motor as required.

Okay, assuming that runStepper does get launched, the rest of the start method takes care of initializing variables used by the object. Another important point to remember is that all Spin code lives in the main system RAM so the background method (runStepper) has access to the global variables in the stepper object. To control the stepper, then, we will create methods that modify the global variables used by the runStepper method.

Since runStepper is king, let's have a look at it.

```
PRI runStepper(m1) | m4, c

  m4 := m1 + 3
  dira[m4..m1]~~

  timer~
  c := cnt
  repeat
    waitcnt(c += clkfreq / 1000)
    if (mode & M_ENABLED) and (mode & M_RUN)
      timer := ++timer // stepTm
      if (timer == 0)
        if (mode & M_STEPS)
          if (numSteps-- > 0)
            if (mode & M_REV)
              stpIdx := (stpIdx + 3) // 4
              outa[m4..m1] := Steps[stpIdx]
            else
              stpIdx := ++stpIdx // 4
              outa[m4..m1] := Steps[stpIdx]
        else
          if (mode & M_REV)
            stpIdx := (stpIdx + 3) // 4
            outa[m4..m1] := Steps[stpIdx]
          else
            stpIdx := ++stpIdx // 4
            outa[m4..m1] := Steps[stpIdx]
```

At first blush this may look a bit complicated, but as you dig in you'll see it's a lot of small, logical blocks. What we have here – in about 30 lines high-level of code – is a background stepper motor driver. I just think that is very cool.

The method starts by making the motor control pins outputs; we discussed the reason for doing that here earlier. The rest of the code is contained with in a big repeat loop that will run until the cog gets unloaded. At the top of the loop is a waitcnt that inserts a one millisecond delay, so this means that all the rest of the code in the loop will get processed once per millisecond.

We have four status bits that affect the stepper. The first two that we test for are the L293D enabled bit and the motor running bit. If either of those bits is zero (off) then there is no point in going any further. Let's assume that the L293D is enabled and the motor is set to run.

If we inserted the pause method from the simple program in the loop we would not be able to affect the stepper on-the-fly, so timing is handled with a global variable using the increment and modulus strategy that we've used so many times in the past (in PBASIC and Spin). By making timer a global variable it can be reset with stepTm when we want to affect the motor speed, and we will get a near-instant change. If timer were a local variable we would have to wait for the current step timing to complete before the change occurred – this might not be a good thing if we had a very long step time.

Once the step timing expires we check the current mode: step (the motor runs numSteps steps and then stops) or free run (motor spins continuously at current speed). When in step mode we check to see if there are any steps left and if so, run a step based on the direction bit. When in free-run mode, we simply look at the direction and do a step. The code should look familiar as we simply copied it from the stepFwd and stepRev methods used in the first program.

You'll notice that there is a bit of redundant code in this method. Again, this method is running in its own cog and has its own I/O definitions. If we attempted to use the stepFwd and stepRev methods from the first program we would have two cogs (foreground and background) attempting to control the same set of pins – not a good idea. Ask me how I know this (yes, I tried….).

Okay, let's let 'er rip. We don't need to go through the entire demo – here's the code that defines the motor and gets it running.

```
OBJ

  motor : "stepper"

PUB main

  if motor.start(EN, MIN_STEP_MS)
    motor.enable(true)
    motor.setRun(true)
    motor.stepRun

    repeat idx from 1 to 5
      motor.setSteps(STEPS_PER_REV)
      repeat until (motor.getSteps == 0)
      pause(200)
      motor.setSteps(-STEPS_PER_REV)
      repeat until (motor.getSteps == 0)
      pause(200)
```

```
    motor.setStepTmr(500 / STEPS_PER_REV)
    motor.freeRun

    motor.setRun(false)
    motor.setDir(motor#RUN_FWD)
    motor.setRun(true)
    pause(5_000)

    motor.setRun(false)
    motor.setDir(motor#RUN_REV)
    motor.setRun(true)
    pause(5_000)

    motor.setRun(false)
    motor.stop
```

We start by defining a stepper object called motor and launching it into its own cog. If that works out, the demo puts the stepper into step mode and spins it back-and-forth five times. Note that when we set the number of steps to a negative value, the motor direction is set to reverse. This strategy should be useful for those doing positioning projects and looking at a change of locations; a positive change moves the motor forward, a negative change moves the motor in reverse.

After the wig-wag loop the step timing is changed to two revolutions per second – we can do this easily because our step timing is specified in milliseconds. After that we put the stepper into free run mode, make sure the direction is forward (note the use of the named constant, RUN_FWD, from the stepper object) and then let it run for five seconds. After the delay the motor is stopped, the direction is reversed and the motor is allowed to run for another five seconds. Finally, the motor is stopped and the object is unloaded.

Well, I think that's about enough fun for this month, don't you? Remember, this stepper object is just a starting point and one could certainly add features to it. Before you do that, though, make sure that you've got a good grasp on how the essential code works; once you do, this program can serve as a template for many other hardware control objects.

Until next time, Happy Spinning!

Resources:

Jon Williams: jwilliams@efx-tek.com

Parallax, Inc.: www.parallax.com

Project Code

```
' ============================================================================
'
'   File..... stepper_demo.spin
'   Purpose... Demonstrates stepper.spin object
'   Author.... Jon Williams, Copyright (C) 2006
'   E-mail.... jwilliams@efx-tek.com
'   Started...
'   Updated... 17 JUN 2006
'
' ============================================================================
'
' Permission granted by author for anyone to use/modify/redistribute this
' code.  No warranty of suitability for your applicationis expressed or
' implied.
'
' Connections (for L293D DIP):
' ------------------------------------
' P0 -> L293D.1 and L293D.9
' P1 -> L293D.2, L293D.3   -> Phase 1
' P2 -> L293D.7, L293D.6   -> Phase 2
' P3 -> L293D.10, L293D.11 -> Phase 3
' P4 -> L293D.15, L293D.14 -> Phase 4


CON

  _CLKMODE      = XTAL1 + PLL16X
  _XINFREQ      = 5_000_000

  EN            = 0                             ' L293 enable inputs (both)
  M1            = EN + 1                        ' motor connections
  M4            = EN + 4                        ' -- (contiguous)

  STEPS_PER_REV = 48                            '
  MIN_STEP_MS   = 5                             ' minimum step time


VAR

  long  idx


OBJ

  motor : "stepper"


PUB main

  if motor.start(EN, MIN_STEP_MS)
    motor.enable(true)                          ' enable driver
    motor.setRun(true)                          ' start the motor
    motor.stepRun                               ' set to step mode

    repeat idx from 1 to 5
```

```
      motor.setSteps(STEPS_PER_REV)             ' one revolution forward
      repeat until (motor.getSteps == 0)
      pause(200)
      motor.setSteps(-STEPS_PER_REV)            ' one revolution reverse
      repeat until (motor.getSteps == 0)
      pause(200)

    motor.setStepTmr(500 / STEPS_PER_REV)       ' set to 1/2 sec per revolution
    motor.freeRun                               ' free run mode when enabled

    motor.setRun(false)                         '
    motor.setDir(motor#RUN_FWD)                 ' set direction to forward
    motor.setRun(true)                          ' start motor
    pause(5_000)                                ' run for 5 seconds

    motor.setRun(false)
    motor.setDir(motor#RUN_REV)
    motor.setRun(true)
    pause(5_000)

    motor.setRun(false)
    motor.stop                                  ' unload object


PRI pause(ms) | c

  c := cnt                                      '
  repeat until (ms-- == 0)                      ' repeat while time left
    waitcnt(c += clkfreq / 1000)                ' wait 1 ms
```

```
' ============================================================================
'
'   File..... simple_stepper.spin
'   Purpose... Simple Stepper Motor Demo
'   Author.... translated (with modifications) by Jon Williams
'   E-mail.... jwilliams@efx-tek.com
'   Started...
'   Updated... 17 JUN 2006
'
' ============================================================================
'
' Connections (for L293D DIP):
' ------------------------------------
' P0 -> L293D.1 and L293D.9
' P1 -> L293D.2, L293D.3   -> Phase 1
' P2 -> L293D.7, L293D.6   -> Phase 2
' P3 -> L293D.10, L293D.11 -> Phase 3
' P4 -> L293D.15, L293D.14 -> Phase 4


CON

  _CLKMODE      = XTAL1 + PLL16X
  _XINFREQ      = 5_000_000
```

```
  EN            = 0                             ' L293D enable pin
  M1            = EN + 1                        ' motor connections
  M4            = M1 + 3                        '  (contiguous)

  STEPS_PER_REV = 48                            ' steps per revolution


VAR

  long  idx, stpIdx, stpDelay


PUB main

  dira[M4..EN]~~                                ' make pins outputs
  outa[EN]~~                                    ' enable the L293D

  repeat
    stpDelay := 5                               ' set step delay
    repeat idx from 1 to STEPS_PER_REV          ' one rev forward
      stepFwd

    pause(500)

    repeat while (stpDelay < 50)                ' modified speed loop
      repeat idx from 1 to STEPS_PER_REV        ' one revolution
        stepRev                                 '   in reverse
      stpDelay := stpDelay * 125 / 100          ' add 25% to step delay

    pause(500)


PRI stepFwd

  stpIdx := ++stpIdx // 4                       ' point to next step
  outa[M4..M1] := Steps[stpIdx]                 ' update outputs
  pause(stpDelay)


PRI stepRev

  stpIdx := (stpIdx + 3) // 4                   ' point to previous step
  outa[M4..M1] := Steps[stpIdx]                 ' update outputs
  pause(stpDelay)


PRI pause(ms) | c

  c := cnt                                      ' sync with system counter
  repeat until (ms-- == 0)                      ' repeat while time left
    waitcnt(c += clkfreq / 1000)                ' wait 1 ms


DAT

  Steps       byte      %0011, %0110, %1100, %1001      ' step table
```

```
' ============================================================================
'
'   File..... stepper.spin
'   Purpose... Stepper motor object
'   Author.... Jon Williams, Copyright (C) 2006
'   E-mail.... jwilliams@efx-tek.com
'   Started...
'   Updated... 18 JUN 2006
'
' ============================================================================
'
' Permission granted by author for anyone to use/modify/redistribute this
' code.  No warranty of suitability for your applicationis expressed or
' implied.

CON

  M_ENABLED = %0001                         ' driver enabled / disabled
  M_RUN     = %0010                         ' run / hold
  M_STEPS   = %0100                         ' do steps / free-run
  M_REV     = %1000                         ' reverse / forward

  RUN_FWD   = 0
  RUN_REV   = 1


VAR

  long  cogon, cog                          ' cog status
  long  stack[64]                           ' local stack

  long  en, mode
  long  minStpTm, timer, stepTm, numSteps, stpIdx


PUB start(ePin, mnTime) : okay

'' Load background stepper controller if cog available

  okay := cogon := (cog := cognew(runStepper(ePin + 1), @stack)) > 0

  if okay
    en := ePin                              ' save enable pin
    minStpTm := stepTm := mnTime            ' save minimum step time
    setMode(%0000)                          ' set defaults
    enable(false)
    stpIdx~                                 ' clear step index
    numSteps~                               ' clear steps


PUB stop

  if cogon~                                 ' if object running, mark stopped
    dira[en]~                               '   float enable pin
    cogstop(cog)                            '   stop the cog
```

```
PUB setMode(m)

  mode := m & %1111                             ' update mode

  if (mode & M_ENABLED)                         ' check L292D enable output
    enable(true)                                '   enable
  else
    enable(false)                               '   disable


PUB getMode

  return mode                                   ' return current mode


PUB enable(status)

  if status
    mode := mode | M_ENABLED                    ' set enable bit
    outa[en]~~                                  ' enable pin high
    dira[en]~~
  else
    mode := mode & !M_ENABLED                   ' clear enable bit
    outa[en]~                                   ' enable pin low
    dira[en]~~


PUB setRun(status)

  if status
    mode := mode | M_RUN                        ' set running bit
  else
    mode := mode & !M_RUN                       ' clear running bit


PUB setSteps(s)

  if (s => 0)                                   ' positive step count?
    numSteps := s                               ' yes
    setDir(RUN_FWD)                             '   run forward
  else
    numSteps := -s                              ' no, make positive
    setDir(RUN_REV)                             '   run in reverse


PUB getSteps

  return numSteps                               ' return curren step count


PUB stepRun

  mode := mode | M_STEPS                        ' set steps bit
```

```
PUB freeRun

  mode := mode & !M_STEPS                     ' clear steps bit


PUB setDir(d)

  if (d == RUN_FWD)
    mode := mode & !M_REV                     ' set to forward
  else
    mode := mode | M_REV                      ' set to reverse


PUB setStepTmr(time)

  stepTm := time #> minStpTm                  ' set new step time (with min)
  timer := stepTm                             ' reset background timing


PUB getStepTmr

  return stepTm


PRI runStepper(m1) | m4, c

'' Run stepper motor as "background" process
'' -- stepper.start method launches this method into separate cog

  m4 := m1 + 3                                ' calculate last pin
  dira[m4..m1]~~                              ' make outputs for this cog

  timer~                                      ' clear timer
  c := cnt                                    ' sync with system counter
  repeat                                      ' run until cog unloaded
    waitcnt(c += clkfreq / 1000)              ' wait 1 ms
    if (mode & M_ENABLED) and (mode & M_RUN)      ' background action active?
      timer := ++timer // stepTm
      if (timer == 0)                         ' ready for step?
        if (mode & M_STEPS)
          if (numSteps-- > 0)                 ' any steps left?
            if (mode & M_REV)                     ' check direction
              stpIdx := (stpIdx + 3) // 4         ' point to previous step
              outa[m4..m1] := Steps[stpIdx]
            else
              stpIdx := ++stpIdx // 4             ' point to next step
              outa[m4..m1] := Steps[stpIdx]
        else
          if (mode & M_REV)                       ' check direction
            stpIdx := (stpIdx + 3) // 4
            outa[m4..m1] := Steps[stpIdx]
          else
            stpIdx := ++stpIdx // 4
            outa[m4..m1] := Steps[stpIdx]
```

```
DAT
  Steps       byte      %0011, %0110, %1100, %1001      ' step table
```