Forthware: Using Forth to manipulate the real world

Stepper Motors

Skip Carter
Taygeta Scientific Inc.

1. Introduction

This is the first in a series of articles on using Forth to interact with the real world. We will explore how to control motors of various types (such as servomotors and stepper motors), switch power to devices, and sense the environment. Each article will present a project that can be used to demonstrate the ideas we are going to discuss.

In this first article, I want to lay the foundation for the future columns and discuss the use of the PC parallel port to control stepper motors. We will adopt the fantasy that we are working on some microprocessor-based control application and will be using the PC parallel port as a proxy for the digital I/O channels on our controller. To the extent possible, the code will be written in high level (so that we can illustrate the principles clearly), and will be in ANS Forth.

2. The PC Parallel Port

First, if you haven't already, go to your back issues of Forth Dimensions and find Ken Merk's article "Forth in Control," (*FD* XVII/2). In that article, Ken talks about using the PC parallel port for eight digital outputs. We will be expanding on that and use some of those other pins to get *input* as well as to provide output.

A parallel port on the PC is really three address locations which, for conventional use, could be called *#Data*, *#Command*, and *#Status*. The port #Data is at the base address of the parallel port, #Status is at the base address plus one, and #Command is at base plus two.

The base address depends upon which parallel port we are using and the hardware installed in your computer; usually, this address is one of the hex addresses 03BC, 0378, or 0278. The BIOS determines the address and maps it to the parallel ports at boot time. This allows an application to find out where the port is by simply reading the table in memory that starts at 0040:0008. Ken shows in his article how to get this value and set a constant containing the base address for the first port; we will do the same here.

Table One. The PC parallel port

DB-25 Pin	Signal	Direction	Port	Bit
1	Strobe*	out	#Command	0
2	Data0	out	#Data	0
3	Data1	out	#Data	1
4	Data2	out	#Data	2
5	Data3	out	#Data	3
6	Data4	out	#Data	4
7	Data5	out	#Data	5

8	Data6	out		#Data	6
9	Data7	out		#Data	7
10	Ack*	in		#Status	6
11	Busy	in		#Status	7
12	Paper	out	in	#Status	5
13	Select	out	in	#Status	4
14	Auto_Feed*	out		#Command	1
15	Error*	in		#Status	3
16	Init*	out		#Command	2
17	Select in*	out		#Command	3
18 to 25	Ground	NA		NA	NA

Table One shows what all the pins on the connector are for. You will notice that #Status port bits zero, one, and two and #Command bits five, six, and seven are not used. The #Command port is used as an output port when the port is being used for a printer, but it is actually an open-collector I/O port and can be used for input. The #Data port latches whatever was written to it, so a read from that port returns the same value that was last written to it. A single PC parallel port then gives us 12 output bits and four input bits, under normal circumstances. (Many PCs use general-purpose parallel I/O chips to implement the parallel port and can actually be programmed to be bi-directional on the pins. Unfortunately, this form of the port is not universal.) For this project we will only need the first four data lines and ground (DB-25 pins two through five and pin 25).

3. Stepper Motors

As our first application, let us consider the control of stepper motors. Stepper motors provide *open-loop, relative* motion control. Open loop means that, when you command the motor to take 42 steps, it provides no direct means of determining that it actually did so. The control is relative, meaning that there is no way to determine the shaft position directly. You can only command the motor to rotate a certain amount clockwise or counter-clockwise from its current position. These "commands" consist of energizing the various motor coils in a particular sequence of patterns. Each pattern causes the motor to move one step. Smooth motion results from presenting the patterns in the proper order.

Features that stepper motors provide include:

- Excellent rotational accuracy
- Large torque
- Small size
- Work well over a range of speeds
- Can be used for motion or position control

There are two types of stepper motors:

- *Bipolar* motors, with *two coils*. These have four wires on them (see Figure One-a). They are tricky to control because they require changing the direction of the current flow through the coils in the proper sequence. We will discuss these motors further when we get to the topic of DC motor control.
- *Unipolar* motors, with *two center-tapped coils* which can be treated as four coils (see Figure One-b). These have six or eight (or sometimes five) wires, and can be controlled from a microprocessor with little more than four transistors (see Figure Two).

Figure One. (a) The internal arrangement of the coils for a bipolar stepper motor. (b) The internal arrangement of the coils for a unipolar stepper motor. Wires a through d are attached to the positive motor power supply. Six-wire motors internally connect a with b and c with d; five-wire motors internally connect a, b, c, and d.

Figure One-a Figure One-b

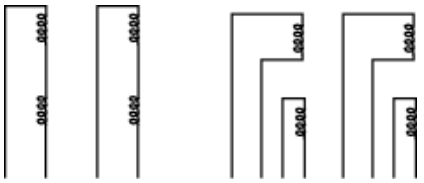
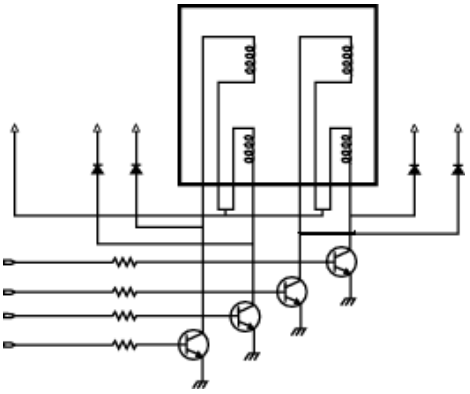


Figure Two. The interface circuit to control unipolar stepper motors from a four-bit I/O port.



Stepper motors vary in the amount of rotation delivered per step. They can turn as little as 0.72 degree to as much as 90 degrees per step. The most common motors are in the 7.5 degrees- to 18 degrees-per-step range. Many have integral reduction gear trains so that they have even higher angular resolution. The motor shaft will freely rotate when none of the coils are energized, but if the last pattern in a series is maintained, the motor will resist being moved to a different position. Because the motors are open-loop, if you do manage to mechanically overwhelm the motor and turn the shaft to a new position, the motor will not try to restore itself to the old position.

There are stepper motor driver ICs available, but these can be *very* expensive (as much as $20 to $50). The sequences are relatively easy to generate with a couple of TTL or CMOS chips at a much lower cost. This is the approach I typically use for most of my real stepper motor applications, since it is a good compromise between parts cost and part count, and it has a low impact on my I/O pin budget.

The easiest way to control a stepper motor is by using four bits of a parallel I/O port from a computer or microprocessor. I usually use this approach when experimenting or when the part count must be as small as possible. The microprocessor approach also has the advantage of being able to use more than one stepper sequence.

The interface to control the motor from the parallel port is just a transistor switch replicated four times. The transistor is there to control the current, which is much higher than the parallel port can sink, and to allow for the motor voltage to be independent of the PC power supply. The circuit in Figure Two can readily work with motor voltages in the range of five to 24 volts. A positive voltage at the transistor base (writing a '1' to the appropriate bit at #Data) causes the transistor to conduct. This has the effect of completing the circuit by hooking up ground to the motor coil (which has a positive voltage on the other side), so the chosen coil is turned on.

Table Two. The parts list.

4	TIP 120	NPN Power Darlington transistors
4	10-K Ohm	1/4 Watt resistors
4	1N4004	Diodes
1	DB-25 Male	solder-type connector

Switching is one of the primary uses of transistors -- we are using a power transistor so that we can switch lots of current (up to five amps for the TIP 120). A Darlington transistor is really a transistor pair in a single package with one transistor driving the other. A control signal on the base is amplified and then drives the second transistor. The resulting circuit can not only switch large currents, but it can do so with a very small controlling current. The resistors are to provide current limiting through the parallel port. The diodes are a feature typical of circuits that handle magnetic coils, that is *inductive* circuits. In this context, the motor windings are the inductive element. Capacitors provide a means for the storage of electrical *charge*, inductors provide a means for storage of electrical *current*. The driving current causes a magnetic field to be built up in the coil. As soon as the drive is removed, the magnetic field collapses and causes the inductor to release its stored current. Semiconductors are particularly sensitive to these currents (they briefly become conductors and then become permanent nonconductors!). The diodes provide a mechanism to safely shunt these currents away and, thus, protect the transistors and the computer. We will be seeing shunting circuits of various types in all the devices we will consider when inductive loads are involved.

The whole circuit can easily be built on a 1 7/8" by 2 3/4" prototyping board. For experimenting, it is convenient to connect the motor to the circuit via one or two feet of hookup wire with alligator clips on them instead of wiring the motor directly to the circuit. That way, a different motor can be attached to the circuit in a few seconds. You should also note that ground for the transistors must be made common between the parallel port (say at pin 25) *and* the motor power supply. An additional wire with an alligator clip can be used to provide access to the ground for the motor power supply. So, on the motor side of the circuit we have six wires, one for each coil, one for ground and one for the motor voltage on the shunt diodes. The motor (positive) voltage supply is provided through the common coils.

After building the circuit, connect the transistors Q1 through Q4 (via their current limiting resistors) to the DB-25 connector pins two through five. When attached to the PC parallel port, the transistors will be controlled by the low four bits of the #Data port. Don't forget the ground wire on pin 25!

3.1 Stepper motor sequencing -- unipolar.

There are several kinds of sequences that can be used to drive stepper motors. The following tables give the most common sequences for energizing the coils. In all cases, the steps are repeated when reaching the end of the table. Following the steps in ascending order drives the motor in one direction, going in descending order drives the motor the other way.

Table Three. The normal sequence.

Step	Q4	Q3	Q2	Q1
1	0	1	0	1
2	1	0	0	1
3	1	0	1	0

4	0	1	1	0

Table Four. The wave drive sequence.

Step	Q4	Q3	Q2	Q1
1	0	0	0	1
2	1	0	0	0
3	0	0	1	0
4	0	1	0	0

Table Four shows what is known as the *wave drive* sequence. This sequence energizes only one coil at a time. For some motors, this sequence gives a smoother motion than the normal sequence.

Table Five. The half-step sequence.

Step	Q4	Q3	Q2	Q1
1	0	1	0	1
2	0	0	0	1
3	1	0	0	1
4	1	0	0	0
5	1	0	1	0
6	0	0	1	0
7	0	1	1	0
8	0	1	0	0

Table Five shows the *half-step* sequence. This sequence interleaves the normal and wave sequences. It *doubles* the angular resolution of the steps, so a 200-step-per-revolution motor now takes 400 steps to complete a revolution.

3.2 The bipolar sequence.

Although we will defer the discussion of bipolar stepper motors, for completeness we present the step sequence here in Table Six. These motors cannot be half-stepped.

Table Six. The bipolar sequence.

Step	C11	C12	C21	C22
1	-V	+V	-V	+V
2	-V	+V	+V	-V
3	+V	-V	+V	-V
4	+V	-V	-V	+V

3.3 Timing issues for stepper motors.

Since steppers are mechanical devices, the timing of the step pulses is important.

The motor must reach the step before the next voltage sequence is applied. If the step rate is too fast, the motor can react in one of several ways:

- it might not move at all, or
- it could vibrate in place, or
- it could rotate erratically, or
- it might rotate *in the opposite direction!*

For very smooth startups, the step rate can be started slow and gradually ramped up to a higher rate. The reverse can be done for smooth stops.

3.4 The control software.

The control code steppers.seq can drive a motor with any of the above unipolar sequences in either direction. The code loads fcontrol.seq, from Ken Merk's article, to find the port and define the words to control the bits on the port. Several other files from the Forth Scientific Library are loaded as well: fpc2ans.seq loads an ANS-like layer on top of F-PC (a true ANS Forth would not need this), fsl-util.seq defines several utility words that are used throughout the Scientific Library, structs.seq loads the data structure words. The data structure sequence is defined to easily manage the sequence of values as defined in the sequence tables given in section 3.1 above. The sequence structures keep track of where in the sequence we are, so that there is no jump in the sequencing if one were to type 7 NORMAL STEPS, stopped to (say) read a sensor, and then continued on with another 7 NORMAL STEPS. This could be done with global variables instead of a data structure. However, the use of a data structure to contain this information is much more natural to extend, if the application were to require several stepper motors, than is the global variable approach.

4. The Future

In upcoming articles, we will be looking at various projects to illustrate the use of Forth to control and measure the real world. Please send your comments, suggestions, and criticisms through *Forth Dimensions* [editor@forth.org] or directly to me at skip@taygeta.com. In the meantime, re-tin those soldering irons!

Skip Carter is a scientific and software consultant. He is the leader of the Forth Scientific Library project, and maintains the system taygeta on the Internet. For details, send e-mail.

The relevant Forth code is available via anonymous FTP. Get these files: STEPPERS.SEQ, ANSI.SEQ, DYNMEM.SEQ, FPC2ANS.SEQ, FSL-UTIL.SEQ, STRUCTS.SEQ, and FCONTROL.SEQ. Equivalent code for **Linux** systems is also available -- send e-mail for details.

This article first appeared in *Forth Dimensions* XVII/5.