Rubberbands and BAILING WIRE

by Jack Buffington

Keeping In Step

How to Drive a Stepper Motor

Often new robotic hobbyists decide that stepper motors are the best way to move their robot and will start asking how to control them. Often they will also have read somewhere about being able to control stepper motors using the parallel port of their computer.

While stepper motors can be used to drive a robot around or position an actuator arm, and you can — with a little additional circuitry — use a parallel port to control stepper motors, neither of these are optimal for most robotic projects. A computer is better suited to high-level processing and, besides drawing a tremendous amount of power, makes for a very expensive replacement for a microcontroller, such as a BASIC Stamp or a PIC, which is better suited for this task. Stepper motors are great devices in some situations and shouldn't be discounted entirely, but a little knowledge will go a long way towards picking the ideal motor for your robot.

Let's look at what a stepper motor is and how it is constructed. A stepper motor is a special type of motor that moves in discrete steps. It can allow you to precisely position something without the need for an encoder to give you feedback about its position.

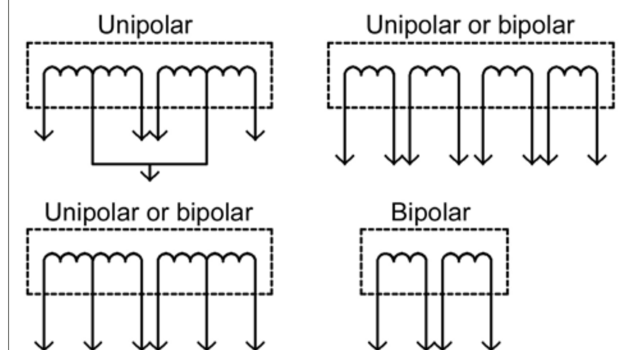
Stepper motors are like permanent-magnet DC motors as far as torque goes. When they are driven at a slow speed, they have plenty of torque. When they are driven at higher speeds, they have less torque. The difference is that while permanent-magnet DC motors have a fairly linear torque curve, the torque for a stepper motor decreases fairly rapidly as the motor speeds up. One additional fact you should be aware of when designing with stepper motors is that they don't take excess torque gracefully. If you pass the rated torque for the speed you are going, the motor will start to lose sync and either fall behind where it is supposed to be or come to a complete stop. This can be a major problem if your robot is likely to encounter a wide range of torque conditions. If you intend to use a stepper motor in your robot, you will need to make sure that you size it so that it can handle any foreseeable amount of torque.

Because of the slipping effect, stepper motors are usually only used in situations where the load on the motor can be predicted ahead of time such as in printers, CNC milling machines, or precise fluid pumping.

Stepper motors come in a wide variety of steps per revolution, but you will find that most stepper motors have 200 or 400 steps per revolution. Stepper motors will have from four to eight wires coming out of them, depending on how they are wound. Usually, they will have four or six wires. Figure 1 shows four different ways that stepper motors can be wired internally. Regardless of the number of wires, all stepper motors operate in a similar manner: you need to energize the coils of the motor in sequence to get it to rotate.

Let's look at Figure 2. In this drawing, you can see a unipolar stepper motor with leads labeled A through D. To the right of the motor are the waveforms that are fed to the motor to get it to turn in the desired direction. The top two patterns use full step patterns. The bottom pattern is what is called half stepping. Half stepping allows you to move the motor through twice as many steps per revolution. There is a third method of driving a stepper motor. This method is called microstepping. Microstepping allows you to position the motor anywhere between the

Figure 1. Four different ways that stepper motors can be wired internally.



NOTE

Last month, I indicated that this month's column would be about using a transceiver from Nordic Semiconductor. In order to provide the readers with the highest quality information, I always make a test circuit and code to ensure that this column is as accurate as possible. There has been some delay with the transceiver circuit, which has prevented it from getting done for this month. Look for information about this circuit in a future column.

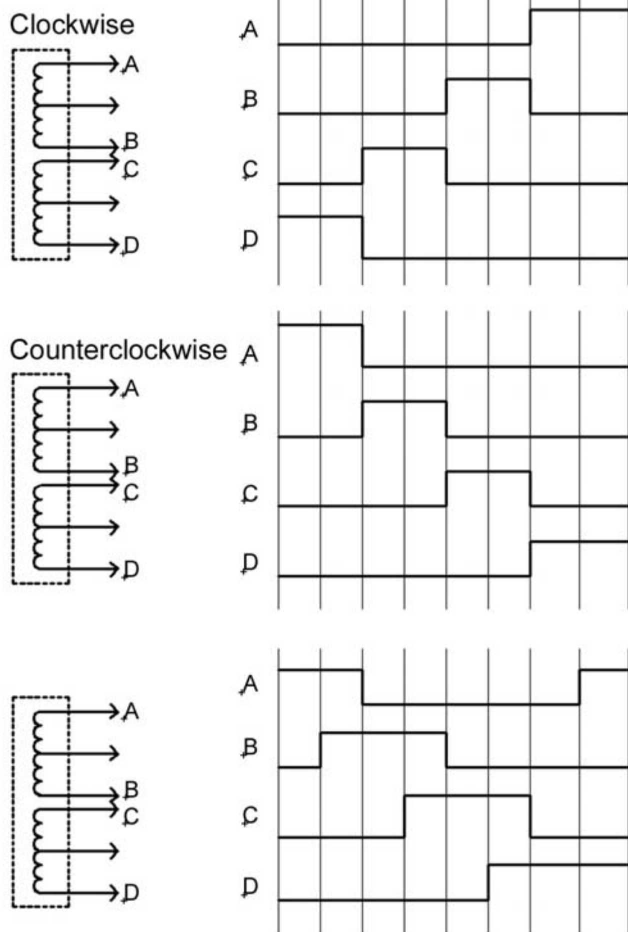


Figure 2. Signals to drive stepper motors.

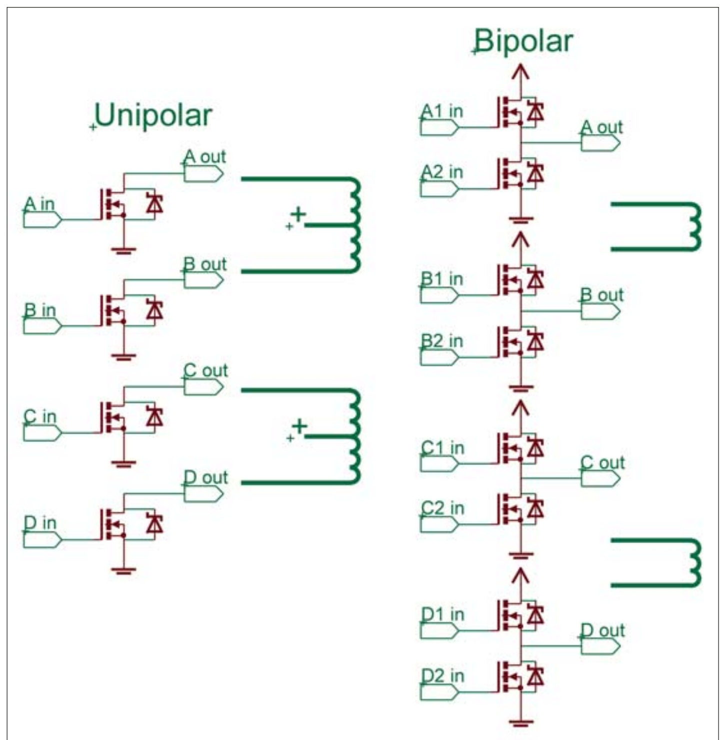


Figure 3. Circuits to drive bipolar and unipolar stepper motors.

whole steps. All you need to do to microstep a stepper motor is to pulse width modulate the leads in a sinusoidal pattern. Be aware, though, that microstepping significantly decreases the amount of torque available.

As you can see, stepper motors are controlled with essentially digital signals. This makes them easy to connect to microcontrollers. Figure 3 shows how to drive unipolar and bipolar stepper motors from a digital circuit. As you can see, the bipolar stepper motor requires two H-bridges to drive the coils in either direction. For the hobbyist, going with a unipolar stepper motor is much easier, though professional applications often use bipolar stepper motors because of their different speed and torque curves.

Let's look now at the software necessary to drive a stepper motor. Without any acceleration, a stepper motor is limited to a small range of speeds. You will need to ramp up the speed of the motor to get it up to its top speed. Most commercially available stepper motor controllers allow you to ramp up the speed of a stepper motor using a trapezoidal speed profile. Let's look at how you can do that using a standard microcontroller.

Ideally, when you are changing the speed of a motor, you would like to make each step a different length. This would give you the smoothest acceleration possible. In practice though, this can be a bit tricky. You could make the duration of each step be some fraction of the previous one. For example, each step could be 254/256ths of the previous step for a slow acceleration or maybe 180/256ths for a fast acceleration. This strategy works but it has a major Achilles heel in that, if you accelerate this way, you aren't guaranteed to have the same number of steps in the acceleration as the deceleration due to the inaccuracy of the math that you are using. Another way would be to calculate through a formula exactly how long each step should take. This would be a great way to do things if you had a lot of processing power available and no other tasks running on the processor.

Unfortunately, in the world of microcontrollers, processing power is somewhat limited, so a third solution to this problem can be used. This strategy uses a lookup table to figure out how long to wait between the pulses. This strategy will allow you to specify how fast the motor accelerates and what the top speed of the motor will be. In this example, Timer 1 on a PIC processor will control the timing of the steps. The clock speed will be 4 MHz, which will divide down internally to one million instructions per second. Timer 1 is a 16-bit timer that counts upwards. When Timer 1 overflows, it will generate an interrupt. If you set the timer to zero every time an interrupt happens, you will get approximately 15 interrupts per second. If you were to set it to 50,000 every time it overflowed, then you would have 64 interrupts per second. To get realistic step rates, you'll need to reset the timer to values around 65,300.

Let's look at how the lookup table that determines the step rate is generated. To get smooth acceleration, your

lookup table will need to have data that graphs out to look like Figure 4. The formula used to arrive at these data points is 65,535 – (range/N). The value of 'range' will determine the slowest speed that the motor can turn. The value used for this column is 5,000. You can figure out what the corresponding step rate would be by dividing your instruction rate by range. In the case of this column, it would be 200 steps per second. This might seem fairly fast, but if you are half stepping your stepper motor and have a 400 step per revolution motor, then it will take four seconds to make a complete revolution.

The next thing that you will need to do is determine the maximum speed that your motor can go. This is limited by two things. The first is how quickly your processor can actually execute its interrupts. In the case of the code presented here, the interrupt can execute in about 70 cycles. This gives approximately 14,300 steps per second. This is a fairly fast step rate. The other limiting factor is how quickly your motor can actually step. Stepper motors max out at a speed that is relatively low compared to permanent-magnet DC motors. Their maximum speed can be as low as 2,500 RPM. The voltage that you are running the motor at will also play a part in determining the maximum step rate of the motor. A good rule of thumb is to start by just using the maximum speed that your processor can handle. You can determine the real maximum speed later if it is lower.

The final variable in the formula is 'N.' This variable transitions from 1.0 to some number that makes the last value in the resulting lookup table be a value that causes the motor to step at its fastest rate. In the case of this column, that value is 65,465 (65,535-70). For this column, the lookup table was generated in Borland C++ Builder, but you could always have your PIC generate the lookup table the first time that it was powered up and store it in its Flash memory. Note that it was chosen to have 128 different speeds that the motor could go. In practice, this is probably more than are needed since you will only notice the speed changes if you accelerate your motor extremely slowly. In most cases you will want to make it spin up to speed as quickly as possible.

Okay! Now you have a nifty lookup table. Let's look at how you can use it. If you were to simply load Timer 1 with each successive lookup table entry each time you stepped, you would be able to get your motor up to speed in 128 steps. In some cases that would be acceptable but why stop there? With a little extra bit of code, you can make the motor accelerate at any rate that you would like. Here is how it is done: You will need to have a 16-bit variable that will help you determine which lookup table value you would like to use. When the motor is stopped, this variable will be zero. Each time the motor steps, you will add an 'acceleration' value to this variable. The upper eight bits of the 16-bit variable will determine which value to use from the lookup table. If your acceleration value is low, it will take a long time to accelerate, if it is high, your motor will accelerate quickly. Decelerating the motor is as simple as taking the same number of steps that you took to accelerate but subtracting the acceleration value from the 16-bit variable each time it steps.

To drive the motor to a new position, you will want to ramp it up in speed, conditionally drive it at a constant rate for a bit, and then ramp it back down to a stop. How many steps will you need to get it up to speed? Let's look at how you can figure that out. First you will need to find how many steps it would take to get to top speed with the current acceleration rate. This is easy enough to do. Simply take how many values are in your lookup table and multiply that number by 256, then divide by your acceleration value. The result will be how many steps it will take you to get to top speed.

If you are limiting your motor's speed to less than its full speed, you will want to reduce the number of steps that it will accelerate and decelerate. To do this, you will use a variable that defines a percentage of the top speed. This value will be from zero to 255. Take the number of steps needed to accelerate to top speed and multiply by your percentage variable. Now divide by 256. This gives you the number of steps it will take to get to the requested speed.

If you just have a short number of steps you would like your motor to make, you will need to limit the number of steps even further. To do this, compare the total number of steps that you will be taking to the number of ramping steps that you have just calculated multiplied by two. You are multiplying by two because you will need to both accelerate and decelerate. If the total number of steps is less than the number of ramping steps, then make the number of your ramping steps be equal to the total number of steps

Figure 4. A graph of a lookup table used to accelerate a stepper motor.

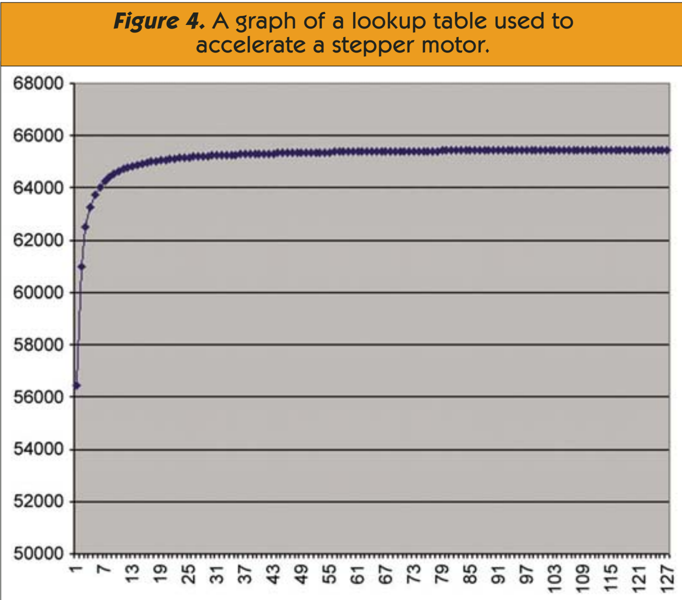


Figure 5. Code to generate a lookup table.

```
const float scaleFactor = .55;
for(int I = 1; I < 128; I ++)
    {
    temp = range;
    Ifloat = I;
    temp /= (Ifloat * scaleFactor);
    temp = 65535 - temp;

    lookup[I] = temp
    }
```

```
int16 calculateRampingSteps(int16 numberOfSteps, int16 accelSpeed, int8 maxSpeedPercent)
   {// Figures out how many steps the motor should accelerate/decelerate for.
   int16 maxPossibleSteps;
   int32 temp32;

   // first figure out how many steps it takes to get to maximum speed ignoring
   // whether that would take the motor past its destination
   // 16384 since only seven bits are used for the lookup table
   maxPossibleSteps = 16384 / accelSpeed;

   // now figure out how many steps it would need to take to get to the top allowed speed
   temp32 = maxPossibleSteps;
   temp32 *= maxSpeedPercent;
   maxPossibleSteps = temp32 / 256;

   // now figure out if the motor can really move that many steps to accelerate
   // and decelerate or if that will be more steps than the move allows.
   if(maxPossibleSteps * 2 > numberOfSteps) // times two because it will decelerate too.
      return numberOfSteps / 2;
   else
      return maxPossibleSteps;
   }
```

Figure 6. Code to figure out how many steps to take for ramping.

divided by two. Take a look at Figure 6 to get a better understanding of how this works. You can find a complete copy of this code on *SERVO*'s website at **www.servomagazine.com**

Finally, you will need to figure out how many steps you will need to take at your top speed. To do this, subtract the number of ramping steps times two from the total number of steps.

You now know how to generate a lookup table, build a circuit to drive stepper motors, and to ramp up and down. The only thing missing is how to tie all of these together. In the example code, you will see that the Timer 1 interrupt is used for the timing of the steps. In the main code, a routine is called that sets up the variables used by the interrupt routine and then enables the Timer 1 interrupt. This routine simply waits until the motor is done making its move and the 'movementDone' variable becomes true before it returns. If your processor has other things to do while moving the motor, you might elect instead to check the 'movementDone' variable before calling the 'moveMotor' subroutine again.

Inside the Timer 1 interrupt, a switch statement is used to determine whether the interrupt should ramp up, run at a constant speed, or ramp down. When the interrupt routine is first enabled, rampSteps, constantSteps, and rampDownSteps have been loaded with how many steps to do in each phase. Each time the interrupt routine is called, rampSteps is decreased by one until it hits zero. At that point, the variable for the switch statement is changed to point execution towards the run-at-constant-speed code. This continues until all of the constant-speed steps have been done. It will then change the switch statement's variable again to point towards the ramp down code.

You are now ready to move your motor! Stepper motors are somewhat complicated to control when compared to permanent-magnet DC motors, but it can be pretty rewarding to see one ramp its speed up and down using code that you wrote. In the right situation, a stepper can provide a much less expensive option for reliable motion control than a permanent-magnet DC servomotor. Your application will determine whether using a stepper motor is a good choice, but if it is, you now have the knowledge to be able to launch into designing for them. **SV**

RESOURCES

Jameco — **www.jameco.com**
Sells stepper motors of various types.

Custom Computer Services, Inc. — **www.ccsinfo.com**
Sells the C compiler that was used with the PIC example code.

Borland — **www.borland.com**
Sells the C compiler that was used with the PC example code.

Microchip — **www.microchip.com**
Manufactures the PIC microcontroller.

STEER WINNING ROBOTS WITHOUT SERVOS!

Perform proportional speed, direction, and steering with only two Radio/Control channels for vehicles using two separate brush-type electric motors mounted right and left with our **mixing RDFR dual speed control**. Used in many successful competitive robots. Single joystick operation: up goes straight ahead, down is reverse. Pure right or left twirls vehicle as motors turn opposite directions. In between stick positions completely proportional. Plugs in like a servo to your Futaba, JR, Hitec, or similar radio. Compatible with gyro steering stabilization. Various volt and amp sizes available. The RDFR47E 55V 75A per motor unit pictured above. **www.vantec.com**

VANTEC

Order at (888) 929-5055