 I got some help with that by looking at the code on the site that I previously posted.  The encoder has 2 channels that output either a HIGH (1) or a LOW (0) signal depending on what it 'sees' on the code wheel (it has very fine lines).  You look for a state change and that tells you that the wheel is rotating.  Based on the the state transitions you can tell what direction you are rotating.  To keep track of the position, a variable gets incremented for a forward transition and decremented for a reverse transition.

For the encoder to actually be of some use, I had to figure out how many lines were on the wheel so that I could compare the potentiometer reading with the encoder reading in my PID controller.  There was no way that I was going to count them manually so I made my arduino do it for me.  The arduino counted every transition as I rotated the wheel for a full revolution.  I did it a couple times to make sure the number I got was correct and then used that number to scale the pot position to a usable number.

<http://olson-projects.blogspot.com/2010/08/position-control-of-dc-motor-part-ii.html>

I researched a little more and I found [PID control](http://en.wikipedia.org/wiki/PID_controller). I read a couple post on the Arduino forums and looked at some code. I found it to be much easier than I thought and implemented it into my code.   
In pseudo code, here is what I added: 

take reading of input pot  
take reading of output pot  
error = input – output  
sum of error += error \* delta time  
error change = (error – last error) / delta time  
motor speed = Kp\*error + Ki\*sum of error + Kd\*error change

The motor speed variable gets translated to a PWM signal that gets sent to the motor to control the speed. The Kp, Ki, Kd are gain constants. They tune the output to get the desired response from the system.

<http://olson-projects.blogspot.com/2010/08/position-control-of-dc-motor-part-1.html>

int val = analogRead(potPin); // read the potentiometer value (0 - 1023)

int target = map(val, 0, 1023, 0, 3600);// set the seek to target by mapping the potentiometer range to the encoder max count

int error = encoder0Pos - target; // find the error term = current position - target

// generalized PID formula

//correction = Kp \* error + Kd \* (error - prevError) + kI \* (sum of errors)

// calculate a motor speed for the current conditions

int motorSpeed = KP \* error;

motorSpeed += KD \* (error - lastError);

motorSpeed += KI \* (sumError);

// set the last and sumerrors for next loop iteration

lastError = error;

sumError += error;

What is going on here is:  
1. read a value from the potentiometer through an analog to digital input (0-1023)  
  
2. map that value so that it lives in the range from 0-3600 (the count from one full rotation of the motor). We need to scale like this so we can make an apples to apples comparison with the current position reported by the optical encoder. The current position is being calculated in another function (an ISR) whenever an interrupt is generated by the optical encoder (more on what this is all about [here](http://www.arduino.cc/playground/Main/RotaryEncoders)).

<http://playground.arduino.cc/Main/RotaryEncoders>

3. create an error term by subtracting the value we want to go to (target) from where we are (encoder0Pos)  
  
4. multiply the error by a constant (proportional gain) to make it large enough to effect the speed. This is the Proportional term, and gives us a simple linear speed value. Basically it says that the further we are from where we want to be, the faster we need to go to get there. The converse is true, so that we will slow down as we approach our destination. Of course, if we are going to fast to begin with, we may overshoot. Then the Proportional will switch signs (from - to + or visa versa) and we will back up towards our target. If the speed is really off then we end up zooming past many times and oscillating until we (hopefully) get to the target.  
  
5. add the difference between the error NOW and the error from the last time around. This is a way to figure out how fast the error is changing, and is the Derivative term of the equation, and you can think of it a bit like looking into the future. If the error rate is changing fast then we are getting close to where we want to be FAST and we should think about slowing down. This helps dampen the oscillation from the position term by trying to slow us down faster if we are approaching the target too quickly. Again, we multiply the value by a constant to amplify it enough to matter.  
  
6. lastly add in the sum of all past errors multiplied by a constant gain. This is the Integral term, and is like looking at our past performance and learning from our mistakes. The further off we were before, the bigger this value will be. If we were going too fast then it will try to slow us, if to slowly, then it speeds us up.  
  
7. set the lastError to the current error, and add the current error to our sum.

<http://abigmagnet.blogspot.com/2008/10/dc-motor-control-part-one.html>

if (encoderPos >= targetHalfway){  
     if (encoderPos >= targetPos){  
            moveInProgress = 0;  
             motorStop();  
        active = 0;  
      }   
      else  if (encoderPos >= targetQuarter){  
        motorSpeed = max(motorSpeed -=10, minSpeed);  
        motorControl(motorSpeed);   
        checkStall();  
            }   
       else {  
         motorSpeed = max(motorSpeed -=5, minSpeed);  
         motorControl(motorSpeed);  
         checkStall();  
       }  
}  
else {  
        motorSpeed = min(motorSpeed +=1, maxSpeed);  
        motorControl(motorSpeed);  
        checkStall();  
}

I suggest you consider using [PID](http://en.wikipedia.org/wiki/PID_controller) to control your motors

The math can be fairly simple:  
  
correction = Kp \* error + Kd \* (error - prevError) + kI \* (sum of errors)  
  
Error is the distance you are from where you want to be.  Correction is the correction you apply to your motor speed based on the error.  It's up to you to choose the appropriate proportional constant Kp, derivative constant Kd, and integral constant Ki for your particular application.  Note that you might be able to get by with just Kp, or maybe just Kp and Kd.  I only very rarely use the integral term, but then again the integral term serves to help the system settle at zero when it otherwise might not from just proportional and derivative alone.  
  
Or you could take a simpler approach of successively refining your position.  Do what you're currently doing, but then afterwards add in a correction stage that moves the motor at a fraction of it's normal speed so that you get closer to your target without overshooting.  I'm not sure if you would have time for this type of correction, but presumably the motor wouldn't have very far to move in the correction stage.

Página que habla sobre los puentes en H

<https://www.pocketmagic.net/a-simple-h-bridge-design/>

**Electronic Components Datasheet Search**

<http://pdf1.alldatasheet.com/datasheet-pdf/view/164530/HP/HEDS-9710.html>

HP components:

<http://www.arbikas.com/view/locator/psc500.pdf>

<http://partsurfer.hp.com/WSearch.aspx?Wchar=Y&searchText=09845>

<http://www.partshere.com/hppictures/>