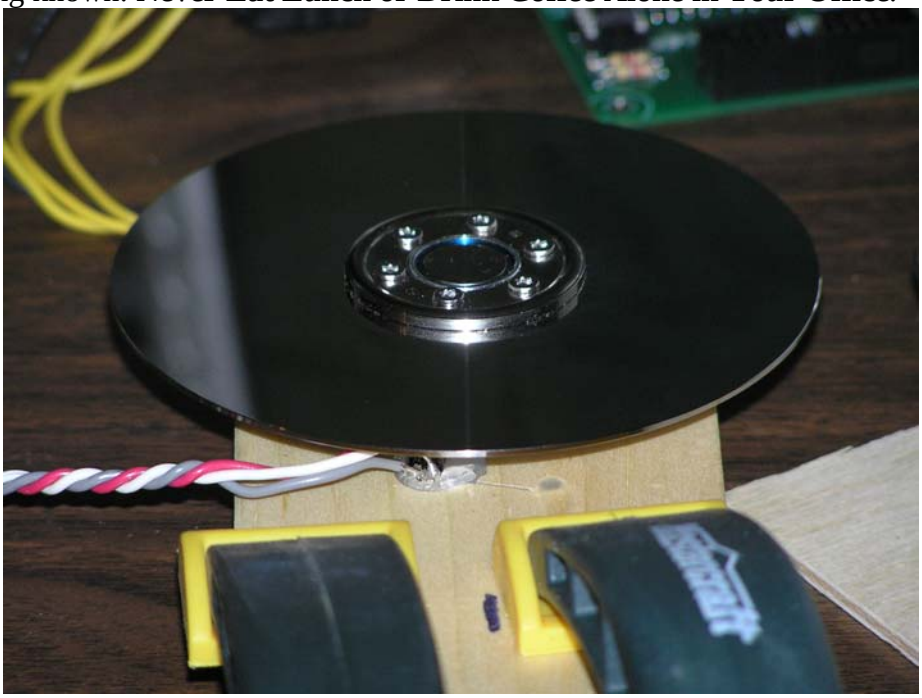
Brushless Motor Driver Project Notes

A note on Serendipity: This successful BLDC motor drive arose because of a chance conversation that Uwe Oehler and I had in the hall way after getting our coffee and before returning to our respective offices. As it turns out, we were both in pursuit of exactly the same goal of driving a hard-disk motor with a microcontroller, although he was using a PIC-based approach and I was working with a Propeller. We had both suffered the same sequence of trials and errors, had attempted pretty much the same methods to get the drive pulses to turn the motor without direct feedback systems, and we were both getting frustrated. Uwe had one more card to play, and that was to add an encoder wheel printed on a plastic film. It was a spectacular success, and I immediately adopted his methods for use on the Propeller. Uwe also developed the speed control approach implemented herein, while I was unsuccessfully trying to use a PID method to control the PWM power delivered to the motor. The take-home message is one that the business world has long known: **Never Eat Lunch or Drink Coffee Alone in Your Office**.

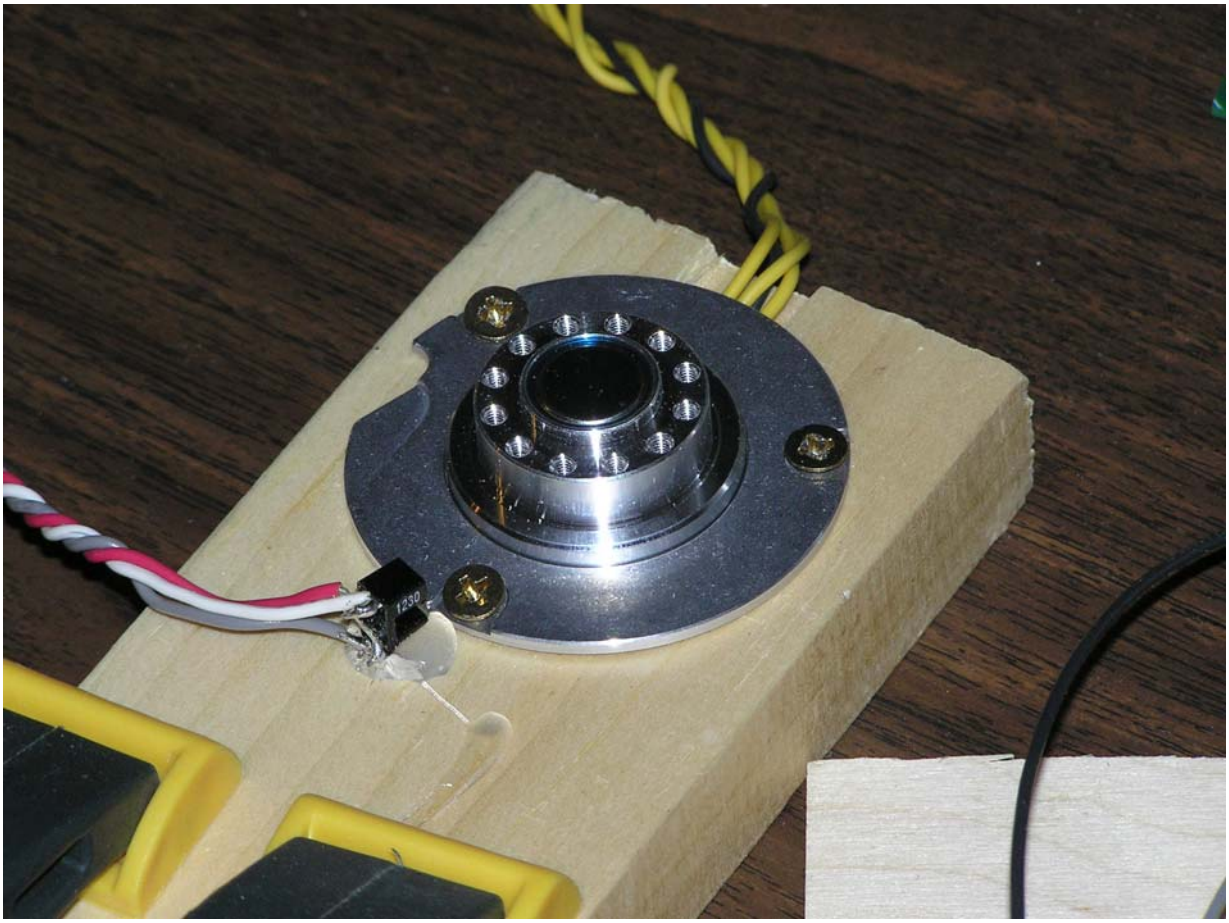


Overview

The following comments may help someone to repeat and improve our implementation of the BLDC motor driver. The goal was to drive a hard-disk drive spindle motor at shaft speeds of at least 100 Hz. These motors are very smooth, very efficient, and provide convenient mounting platforms on which to mount objects such as wheels or chopper disks. Feel free to improvise with the components shown herein. The overall principle is that the activation of individual coils within the motor must done with knowledge of the current rotor position. This information is provided by attaching an optical readout to the rotor. The system is synchronized from standstill and keeps track of the position using hardware counters and dedicated Spin code. Speed control is provided by not activating coils if the speed is too high. Power control is available using pulse width modulation of the drive sequence. In practice, the program parameters are fairly flexible and easy to adjust.

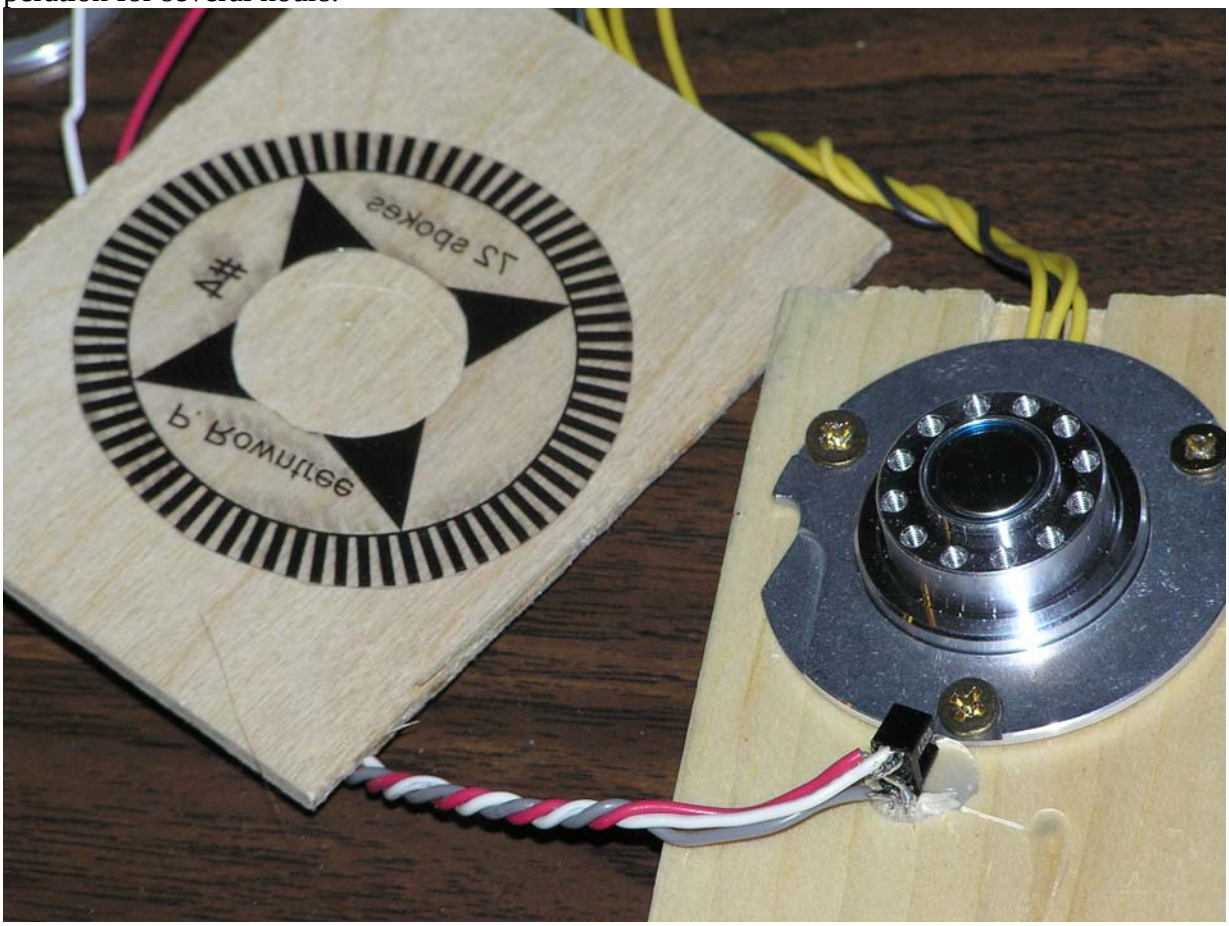
Hardware

The motor was pulled from an HDD. It has three windings, herein referred to as A-B-C, which are repeated 4x each within the body of the motor. Some high-speed motors from SCSI drives have 3 sets of A-B-C windings per rotation. The code should run without modification on these devices, although the ConfigureMotor routine should be given the correct values in order to report correct information. The contacts for the motor are a common and the three leads to these three windings. The resistance of each coil was about 3 Ohms. The higher speed motors often have very low-resistances (~1 Ohm), and the drive circuitry (and power supplies) may have to be adapted for these motors.

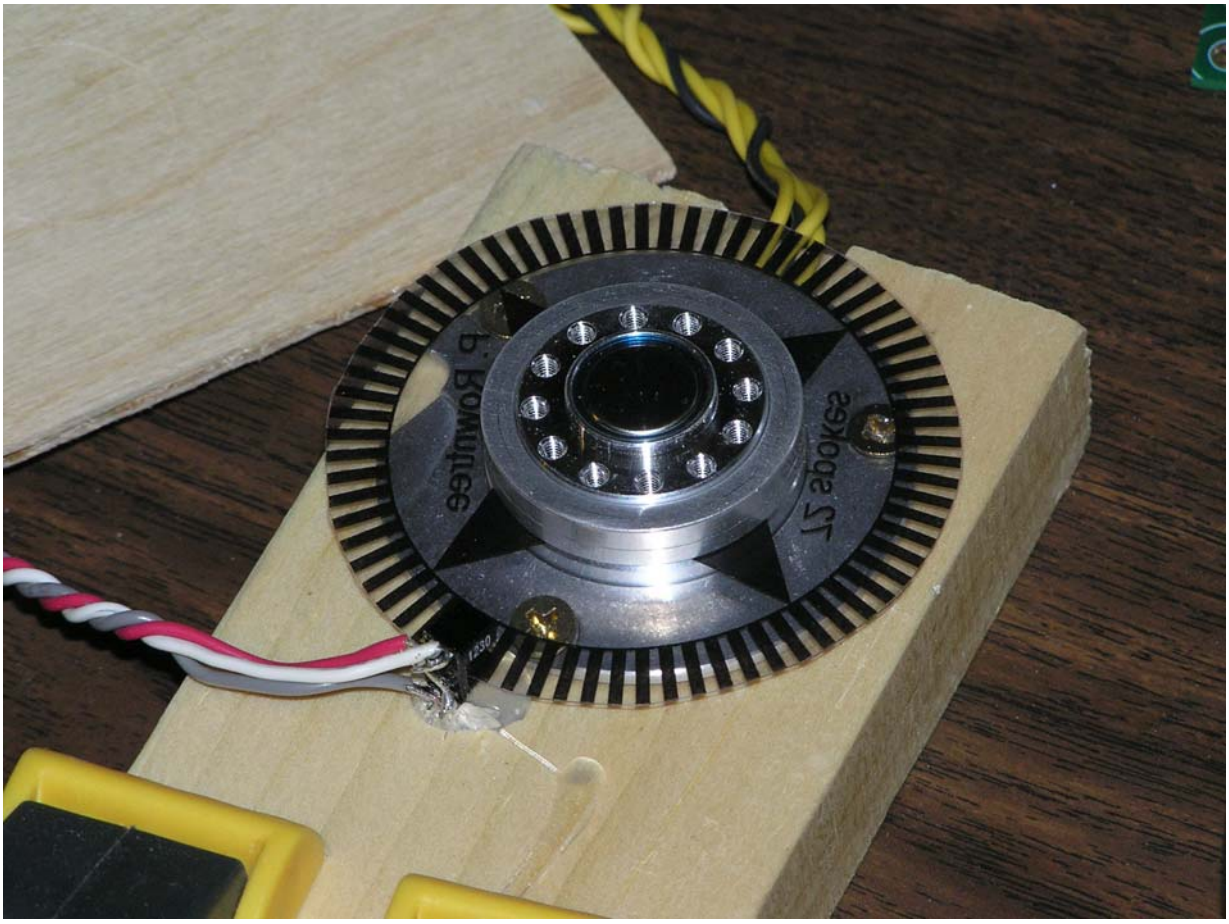


Initial attempts to drive the motor consumed far too much current, and as a result I use heavy-duty TIP-121 Darlington NPN driver transistors. The schematic diagram in the PDF file shows the wiring details. They have a rated capacity of ~8 Amps; in the present design the instantaneous current is ~4 Amps in each coil, but the pulse-width modulation and controlled cycle duration provided in software, combined with the non-overlapping excitation sequences, reduces this to an average current of ~0.5 Amps per coil. This is comparable to what most hard disks draw from the 12 Vdc supply, so we are in the right territory here. A ULN2803A Darlington driver array might work, especially if 2 channels where used for each drive coil. The 12V power supply must be capable of driving this (and a bit more perhaps for highest speed operation). At full power the

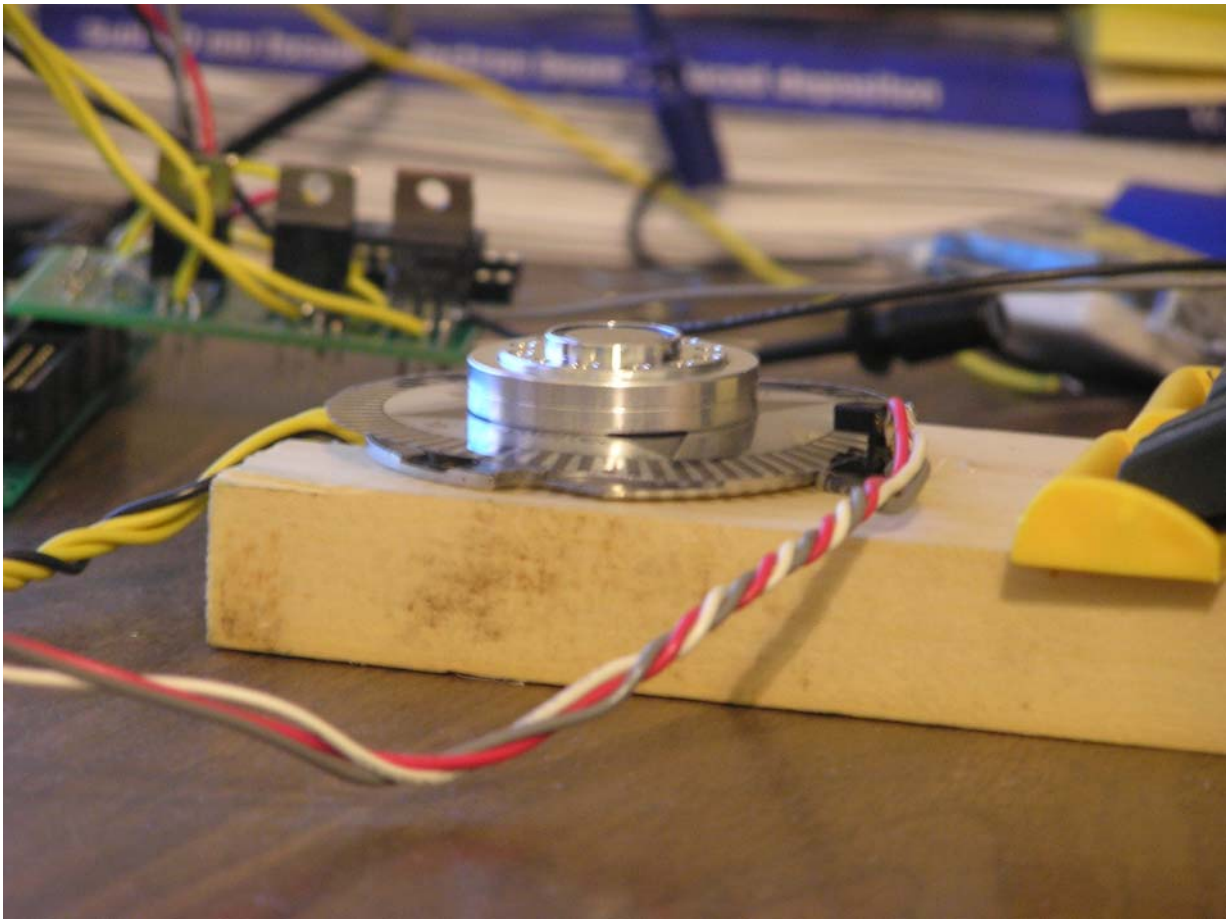
transistors do not get warm even without heat sinks, and the motor only warms slightly after operation for several hours.



The rotation of the motor is detected using a transparent disk which has been patterned using a laser printer. This disk has 72 lines arranged uniformly around the perimeter. A PDF file and a CorelDraw file are included in this package that show several disks that have been used. At the present time I am using a 2.5" diameter disk that is mounted below the main rotor disk. With 72 sectors and 4 sets of A-B-C coils, each coil activation lasts for the passage of 6 sectors. These are monitored using an opto-interrupter. A Vishay device (TCST1230) was used (seen in the above photograph as the black device near the motor mount), but there are many others that are capable of this task; apparently some devices include Schmitt trigger-conditioned outputs, which would be ideal. Just watch that they can switch at least at 10-20 kHz to keep up with the full speed rotation. Components used (see the schematic) are not critical. Resistors R1 is provided to current limit the emitter, while R2/R3 provide a current limit through the detector and a voltage divider to ~3.3 V. Because the raw output of this divider had significant numbers of 'bounce-like' transitions per passage of a single spoke (hundreds to thousands, especially at slow speed), this signal was run through a Schmitt trigger/inverter. The output of the inverter was fed into the Propeller chip, which used counter B of the driver cog to monitor the rotations.



All components are mounted on a single SpinStudio prototyping board, and connect to the SpinStudio motherboard via the B-Socket. As can be seen in the photographs, the motor is mounted to a wooden base, and the opto-interrupter is hot-glued to this same base. The spoked disk is at the lowest level of the motor hub, with the spinning disk near the top of the stack, adjacent to the six bolts that secure this to the hub by downward pressure. I have never found that the absolute alignment of the spoked disk with the internal coil orientation of the motor to be required, presumably because with 72 sectors on the disk there is bound to be an acceptable alignment found by the motor itself, which will 'pull' its operation to the activated coil.



Software

The Spin software is simple. This is enough to get to ~120 Hz every time, and 125 Hz with rare problems; the ultimate speed limit in Spin is ~130 Hz. As written, the system must be started from a stationary rotor. The code gradually increases the average C coil current to push the rotor to a C coil without overshooting. The C coil is de-energiezed, some housekeeping sets up the hardware counters, then the driver solidly kicks the A coil to move the rotor. This configuration provides a reliable way to get the motor going in one direction every time. Simple tweaks to the code will allow the opposite rotation direction (i.e. interchanging any two of the three drive pin definitions). The code then enters the main loop whereby it uses the hardware counter B to count sectors, while applying power to the appropriate coil. Since the code could easily find itself in the *next* coil's space while driving a coil, the code as written will only apply a PWM controlled pulse to the motor winding if less than (Sectors_per_Coil-DeadSectors) of the Sectors_per_Coil associated with a coil have passed. This also reduces the overall power requirements to the motor, and do not seem to affect its top speed of the Spin implementation. My impression is that using a DeadSectors of 3 gives more stable speed control.

The system is setup ***once before running the Start_v1 routine***, which the gets the motor moving. The three configuration routines are

ConfigureEncoder(PinNumber) : Tells the code which I/O pin to treat as an input to read the opto-interrupter signal.
ConfigureMotor(Coils_per_Rotation, Sectors_per_Coil, DeadSectors) : Describes the internal structure of the motor and our driving method for the code. My disks have 72 sectors per rotation, and the motor has 4 cycles of A-B-C coils per rotation. This gives 6 sectors per coil. I find that 3 DeadSectors are useful, so my call is ConfigureMotor(4,6,3)
ConfigureDrive(apin, bpin, cpin, tpin) : tells the driver which pins to manipulate to control the A,B and C coils. The tpin is used to provide a reference pulse of ~50 microseconds at the start of each A-B-C cycle, which is useful for triggering a scope or external monitoring devices.

Once the Start routine has been run (which uses one dedicated cog), the main program is free to adjust the speed and power delivered to the system via
SetControlLevel(Percent, SetPoint): Percent is a 0-100 value to control the duty cycle of the applied power, and SetPoint is the shaft frequency, in Hz, that you wish to obtain.

You can monitor the behaviour of the motor using the 4 Getxxx routines.
GetOnState : returns a TRUE (-1) or FALSE (0) value according to the activation state of the current coil. It will be false if the instantaneous speed is above the setpoint. ViewPort can track this value and plot it to see the long-term behaviour of the activity, and report the average value over a longer period of time.
GetCycleTime : reports the number of clock tics for the most recently completed A-B-C cycle sequence. To avoid jitter, there is a logarithmic filter applied to this value in the code.
GetCoilTime : reports the number of clock tics for the most recently completed coil. To avoid jitter, there is a logarithmic filter applied to this value in the code.
GetFreq : reports the current shaft rotational frequency, in Hz. This is derived from the Cycle time, but is presented in more useful units than clock tics.

The power is delivered to the motor by a PWM approach. Empirically, pulses below ~100 microsecond do not drive the motor well, but this is probably motor dependent. The parameter referred to as 'PulseWidth' (PW) controls the duty cycle of the PWM drives. I find that 60% can get the motor up nearly to 120 Hz, and control it down to ~10 Hz. This is the default PW provided in the code. Higher PW may be required for full speed operation. Note that for the Spin code given here, the parameter of PW is divided by 4 to give a value from 0-25; this is the On time of the device. The Off time is 25-OnTime. As a result, there will not be different behaviours with a PW of 49 to 51 for example. This gave applied pulses that worked well. If a PASM version is written, a more precise approach may be taken.

Speed control is provided by simply monitoring the time taken for a single coil to pass by (according to the sector counter) as well as a complete A-B-C cycle. If this time is below a threshold-specific value, the next set of coil activations are skipped, and the system will naturally spin down until the speed is acceptable, and power is restored on the next coil activation. Speed control being applied to between 20-90% of the coil cycles works well. The present code provides a stable shaft speed with ~±0.5 Hz fluctuations under most conditions. If the requested speed cannot be obtained with the chosen PW, every coil will be activated on every rotation. If the requested speed is too high for the spin driver (~125-130 Hz seems to be the limit for my system) then the code can miss a spoke, and the motor will slow down because the drive signal is

not ideally synchronized with the motor rotations. The cog needs to be restarted to regain optimal control. A PASM driver would certainly avoid this problem..

The code is written to be monitored using Hanno Sander's excellent ViewPort code. The most recent version is 4.1.1; the Conduit module is included in the Spin archive, but has been relabelled from Hanno's filename to avoid confusion or over-writing of important information. If you do not wish to use this interface, then remove the object referred to as 'vp', and remove all lines associated with the ViewPort Conduit cog. The top-level Spin code periodically requests data from the BLDC driver object, and stores them in variables that ViewPort can access and report back to the PC via the Conduit cog. The key variables that the Conduit shares with the PC are:

SetPoint, (expressed in Hz) triggers the speed control mechanism. Look into the code and determine which speed control method (CycleTime vs CoilTime) you wish to apply.
CycleTime (expressed in Propeller clock counts). With the present 4-Cycles/rotation motor, a 200_00 cycle time should provide a ~100 Hz shaft speed.
CoilTime (expressed in Propeller clock counts). With the present 4 Cycles/rotation motor, a 66_666 cycle time should provide a 100 Hz shaft speed.
Freq (expressed in Hz) the current rotational Shaft speed
PulseWidth (PW) to apply to the motor, expressed as a 0-100 percentage.
On TRUE (-1) when the coils are driven, and FALSE (0) when the speed control algorithm is trying to slow the motor down. The ViewPort program can plot this signal and report the average On time for the coil activation.

Performance

These values are going to depend enormously on the motor used, but here are the maximum rotation speeds as a function of the PW parameter. They were obtained using ViewPort, setting the SetPoint to some absurdly small value to disable speed control. This is for a 4-coil set motor, using 6 spokes per coil, and a DeadSectors of 2 spokes at the end of each coil activation.

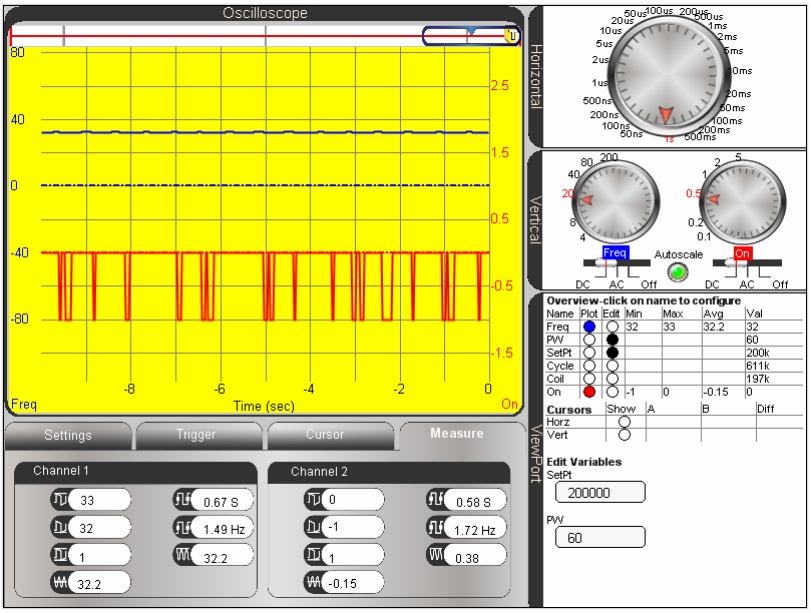
PulseWidth (%)	**Max Rotational Speed (Hz)**
32	73 ±2
40	81 ±2
52	111 ± 3
60	117 ± 4
72	full speed caused spin to loose spokes

Another point of view is to impose a SetPoint and measure how often the motor is driving the coils.

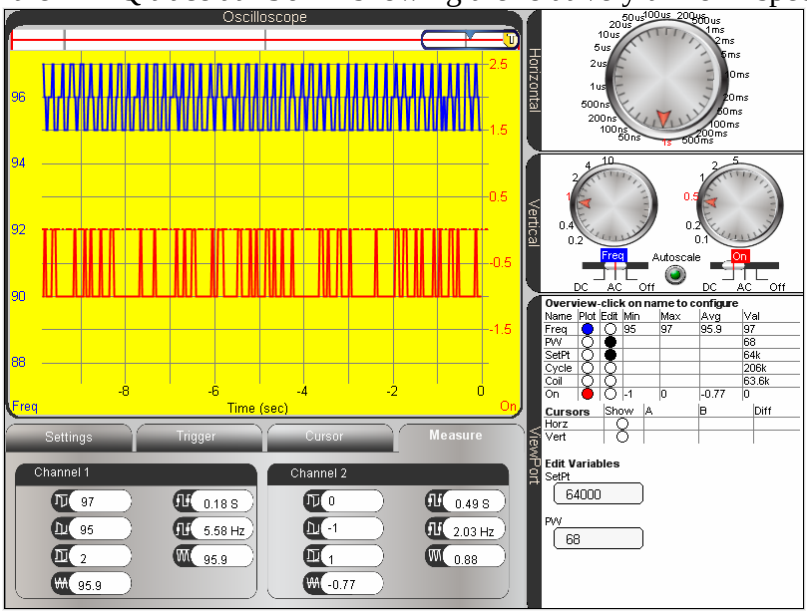
PulseWidth(%)	**Coil Time SetPoint**	**Approx. % Activated Coils**
60	200000 (~32 Hz)	~15 %
	100000 (~63 Hz)	25
	65000 (~95 Hz)	55
52	200000	17
	100000	40
	65000	75
40	200000	35
	100000	55
	65000	100 (unable to achieve setpoint)

ScreenShots of Motor Performance

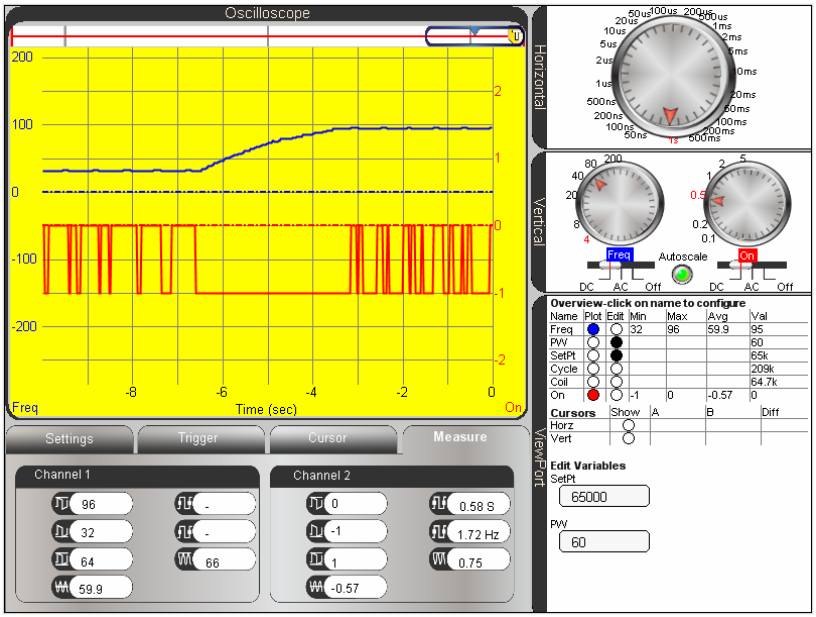
The following is a screen-shot of the ViewPort control program, with the motor gently ticking over at ~32 shaft Hz, using a high PW value of 60%, and then choking it back with a slow speed. The RED curve is the On/Off coil activation signal used for speed control, while the BLUE trace shows the measured FREQ rotational speed. The motor will go as low as 5 Hz, but it is at the low end of stability. **(NOTE : These screenshots were taken of a version that used clock-counts as the SetPoint variable; the provided version uses shaft frequency for convenience)**



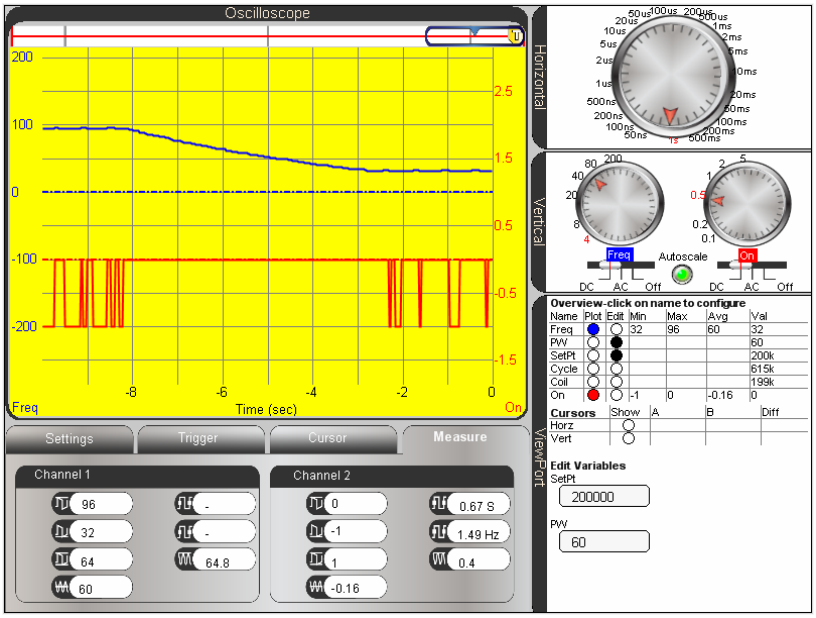
Here is a zoom of the FREQ trace at ~96 Hz showing the relatively uniform speed.



The following is showing the speed up process from the above conditions to the ~100 Hz required for my application; this is achieved by providing a new SetPt value to ViewPort. It takes about 3 seconds for the motor to make the transition, during which time the motor coils are being activated at each opportunity, although still at the PulseWidth of 60% each time. The system comes to speed, then the speed control kicks in, causing the On to start fluctuating again, but clearly spending more time at the logical 'TRUE' value of -1 than it did at the slower rotational speed.



Deceleration is slower, taking about 5 seconds to return to the original 32 Hz shaft speed, as shown below.



Other Considerations

1) If you have difficulty getting the motor to spin up, try investigating the effect of the StartPulseLength parameter in the PositionToCoil( ) routine of the driver code. Longer StartPulseLength will give a stronger kick. I have not required this to be finely tuned, but your mileage may vary. You can also play with the DeadSectors parameter to the ConfigureMotor routine; I use 2, preferably 3, but if you want to live dangerously, you could try 1. The system will un-synch more easily, and this may not be just as the motor speed approaches the top end. Again, a PASM routine would solve this issue.
2) When designing the encoder wheels, make sure that you have good separation of the On/Off cycles, which will depend on the opto-interrupter device chosen, the position of the lines with respect to the centre, and the line widths. Use a laser printer or photocopier that gives nice crisp patterns.
3) Stable mechanical construction is important.
4) If there is any rubbing between the encoder wheel and the opto-interrupter (or anything else for that matter) then the toner markings can rub off enough to cause the optical signal to miss a spoke. This will cause problems that are not apparent, and a hardware system that worked perfectly one day can fail the next. These disks flatten out as they run at higher speeds, so a stable geometry at high speed may rub at slow speeds (or vice versa) if the disk is slightly distorted (as mine tend to be). Better alignment is the best solution, but flipping the encoder wheel to have the toner side avoiding the rub point is another (repetitive scratching of the disk may also cause problems). The best approach is to have as little of the transparency film exposed as possible by providing an upper and lower support disk out to a radius that is close to the opto-interrupter; this will stiffen it considerably.
5) Reasonable improvements would include (a) a PASM driver for higher speed operation, and (b) Developing a way to determine the absolute position of the encoder wheel, at least once per rotation, might allow the system to re-synch itself if problems arose.
6) Don't try to drive the motor with the raw power input to the Propeller boards, unless you are using a very solid supply and large capacitors on the supply lines.
7) The absolute orientation of the 72-spoked encoder disks with respect to the motor spindle is not critical, but if you make wheels with fewer spokes it may become an issue.

Summary

This motor driver seems to be very stable, and as implemented can drive HDD motors at anywhere from 10-120 Hz with no problems. Higher speeds will require PASM drivers.
Good luck, and we hope that others will find this code useful as-is, or as a starting point for further refinements.

Cheers!

Uwe Oehler

Paul Rowntree
Rowntree@uoguelph.ca