-- MECHANICALCAT.ORG --

ADDING AN ARDUINO PID TO A 4QD PORTER-10 DC MOTOR CONTROLLER

Contents

Introduction

This doco briefly covers how to wrap a closed-loop manager around an open-loop DC motor controller so that the user's throttle will select a (maximum allowed) *current* rather than a given *PWM duty cycle*.

Such a controller is of benefit in power-limited systems - for example a Hacky Racer in which the maximum allowed instantaneous power is dictated by an inline battery fuse.

Software and hardware are covered.

Warning

Don't use this in any safety-critical application. See the license terms!

Beware that a typical 1500W brushed permanent-magnet mobility-scooter motor is not a thing to be trifled with. It is extremely heavy and - if powered up without being securely anchored to a substantial workbench - its start-up torque can rip it off the bench and lob it around the worktop with dangerous momentum. Brushless motor controllers generally enforce fairly modest accelerations on their motors, which keeps start-up torque to manageable levels. Brushed motors do *not* play nicely at all and can go from nothing to flat-out in a fraction of a second when power is applied. If you're used to brushless motors then the fierceness of a brushed motor may take you by surprise.

Overview

Core

The core controller - in this case a 4QD Porter-10 - is an open-loop device which converts an input signal into a high-current PWM output for (say) a large DC motor.

In the default open-loop set-up of a Porter-10, the user's input (0-100%) simply* instructs the controller to provide a matching output (0-100% PWM).

**In fact, the Porter-10 is not quite that simple and it does not directly copy the input to the output - internal logic manages the maximum rate-of-change of the output, as well as limiting the peak current to protect the controller output stages against overload. So there is some lag and some peak-cropping between the user's input and the actual output.*

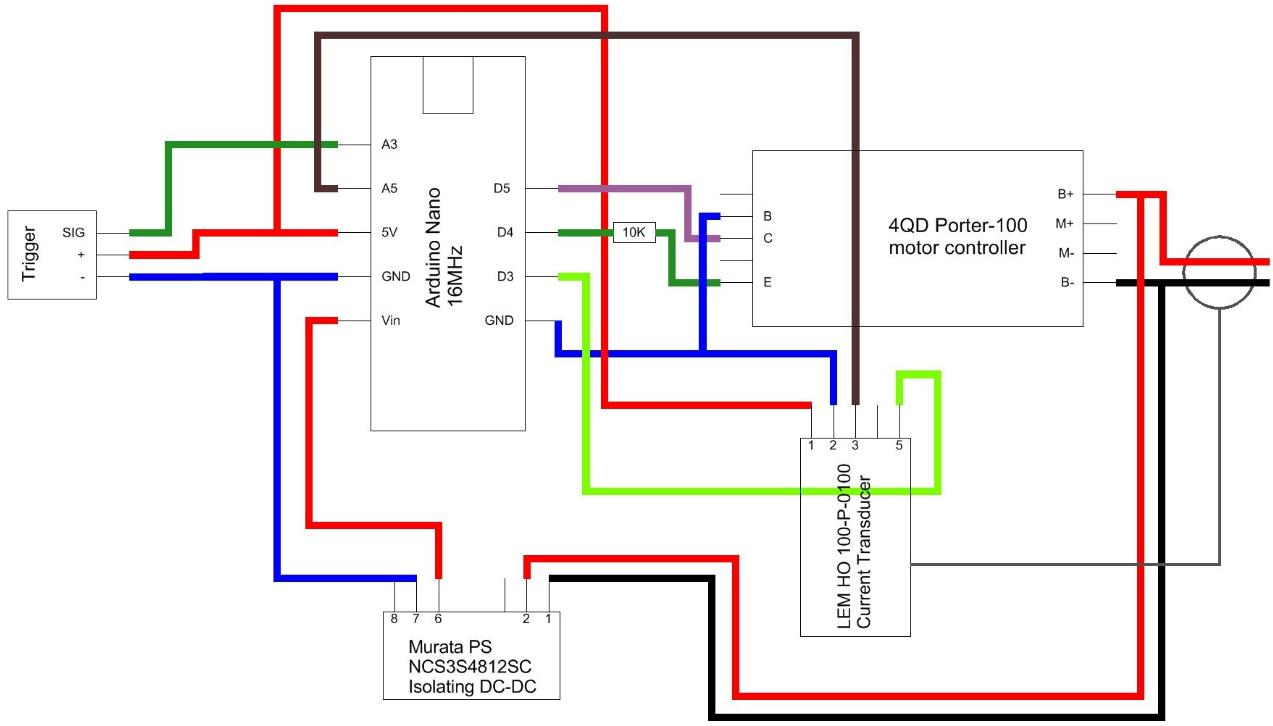
In a system which is *not* power-limited (i.e. there is effectively no upper limit on the permitted current other than the controller's own limitations) then a Porter-10 is a sufficient controller without needing the closed-loop additions documented here.

PID Wrapper

The additional closed-loop wrapper consists of two parts: a current sensor in the battery wires; and an Arduino which monitors the current. The user's input to the system is now via the Arduino, which converts that input to a demanded *current* in a fixed range (typically 0 to 30A) rather than a demanded *duty cycle*. The processor implements a PID feedback loop to match the actual battery current to the demanded current. Sort of :-)

Circuit

Here's a basic circuit diagram.



Main Components:

- 4QD Porter-10 Motor controller. Steve at www.4QD.co.uk takes an interest in Hacky Racers, so let him know what it's for if buying. Other controllers *may* work.
- LEM HO-100-P-0100. +/-250A open loop current sensor. uk.rs-online.com stock number 138-3471. Note that the stock picture is wrong - item is blue.
- Murata Power Solutions NCS3S4812SC 48V-to-12V isolated 3W DC-DC converter. uk.rs-online.com stock number 876-2466.
- Resistor 10Kohm. Any metal-oxide resistor or similar, say 1/8W, should be fine.
- Trigger. Most ebay-supplied 'e-scooter' hall-effect thumb triggers are suitable. A 5Kohm or 10Kohm potentiometer trigger would also work, with some software adjustments. Some (rare) triggers have reversed logic (i.e. ~4V at low throttle, ~1V at high throttle) in which case a software adjustment should be made to match.
- Arduino Nano. What's there to say? Couple of £ from ebay.

Circuit Notes

IMPORTANT: beware that all the Porter-10's control inputs (A/B/C/D/E) must be isolated from the main power supply and from the motor. Therefore there must be **no common ground** between the main battery (B+/B-) and the Arduino's power (Vin/5v/GND). It is essential, therefore, to use an **isolating** DC-DC converter to supply the Arduino and it is essential to leave the Arduino GND floating.

Observe that the Arduino GND is connected to the Porter-10's 'B' ground terminal - not to the 'A' terminal as documented by 4QD. We find this to be more reliable.

The 'Trigger' should be a Hall-sensor throttle (output 0.8V to 4.2V typically) or a 5K/10K potentiometer (giving 0V to 5V). Adjust software to match.

The Arduino's 5V PSU has - to date - proven adequate to supply both the current sensor and the trigger as shown. If prefered, a separate common-ground regulator could be installed on the isolated 12V output to supply these.

Beware that swapping the 12V supply for an isolating 5V supply (wired direct to the Arduino 5V input) is inadequate and results in frequent brown-out reboots of the Arduino irrespective of any capacitative smoothing employed.

Other Core Controllers

The same circuit can be employed on cheap 'Chinese' motor drivers. These are typically 1-quadrant devices, cost about £10-£20, and often come with a potentimeter to select the speed and a physical switch to select on/off. Some even have a reverse implemented by two large relays (ick). Typically the on-off switch is non-logic i.e. it controls the full battery voltage (e.g. 48V) supply into whatever regulator supplies the logic circuitry.

So in general the wiring will need to be changed in the following ways:

- All grounds can - and should - be commoned. But check the circuit to be sure, before doing this. Quite possibly the expensive DC-DC isolating converter could then also be replaced by a simpler 3-wire regulator - although 48V-capable regulators are not commonplace. TI's LM317HVT, perhaps?
- The on/off switch should either be operated by a relay/transistor/FET (via the Arduino's D4 output) or - our preference - should simply be soldered closed.
- The potentiometer should be removed and the Arduino's D5 output connected to the pin on the controller which was previously connected to the pot's wiper. A 10Kohm resistor should be fitted to pull this pin to ground, otherwise the motor will run at random (but dangerously non-trivial) power during boot-up.

In addition software changes may be required:

- Reversing the logic of the D4 output.
- Setting the D5 output to range from 0% to 100% to emulate the full range of a pot.

Further software enhancement - in the case of a reversing controller with relays - would be to enable the reverse changeover only when the throttle and the current draw are at zero. Perhaps with a time delay to prevent driving the motor (say) in reverse whilst carrying forwards momentum - which would cause a very large current surge probably destroying the controller. Further reverse protection could be applied by limiting the maximum reverse PWM to - say - 20%.

Choice of Controller

So far - touchwood - the Porter-10 variant has proved highly reliable. But the existing software has proven inadequate to protect a cheap/dumb 'Chinese' controller against surges, and we have blown up a few of those. You have been warned. A Porter-5 - on paper - would be sufficient for most LiPo 12S Hacky Racer implementations and is slightly cheaper than the Porter-10.

Controller Protection

Most controllers with a FET output stage rely on the presence of the battery to safely 'close' the circuit between the input terminals. If the input suddenly goes open-circuit while the motor is still turning, the controller can be destroyed instantly due to it having nowhere to dump the flyback current from the motor.

On a Hacky Racer (and on most other sensibly-wired devices) there are two ways that the input can go open-circuit:

1. The fuse blows. This invariably happens while the motor is either under heavy load, or under heavy regen braking, because that's when the current is flowing. Either way, the controller is very likely to be destroyed by this event.

2. The main power switch / emergency stop switch is turned off. On a Hacky this is quite likely to happen under full power after (say) the throttle jams open. Again, the controller is very likely to be destroyed by this event too.

Solution

Fit flyback diodes across the switch and the fuse. The diodes should be rated to very high currents (double the fuse rating at least) and should - obviously - be wired so that they conduct in the reverse direction - i.e. to allow flow of current into the battery in the sense of recharging the battery, not discharging it.

Clearly, after wiring the flyback diodes, the circuit should be tested to prove that the diodes are the correct way round. To test:

- Remove the fuse and turn on the main switch: the controller should not power-up.

- Turn off the main switch and refit the fuse: the controller should not power-up.

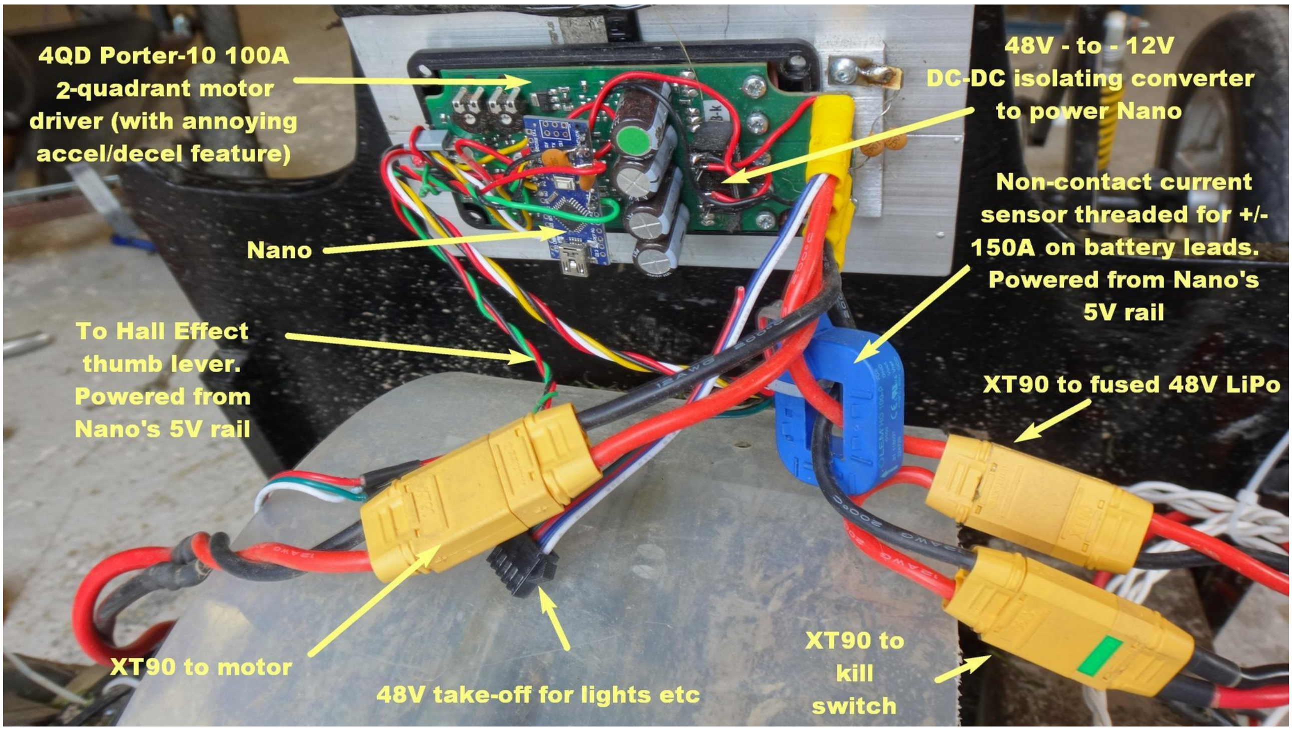
It takes only one flyback event to destroy your controller. Which means it can be toast the very first time the fuse blows. And it's likely that you'll blow fuses while testing and hacking this software and while turing the Porter. So fit the diodes before powering-up anything.

Vishay diode VS-80EBU02 looks a pretty good option for a flyback diode in most 48V/30A Hacky applications. It has an 80A/200V rating and its 'power tab' construction means that it effectively has a mini-heat-sink built into both terminals - which looks like plenty enough for the couple of seconds of use it might get when the fuse blows at max regen. So just solder a (heavy) wire to each end, cover all-over in heat-shrink, and fit. Only £2 each from RS Components. Their item number: 395-3985.

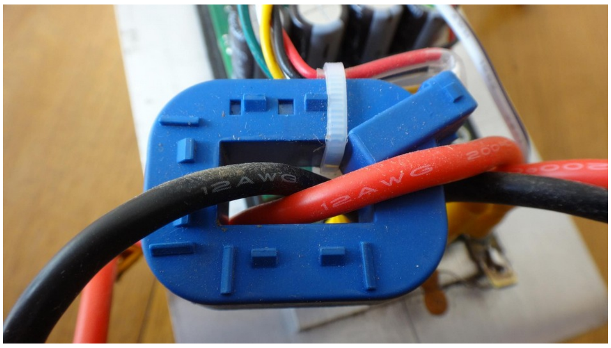
Assembly

The notes here assume you're using a Porter-10. Notes on wiring cheap 'Chinese' controllers will - hopefully - follow in a few weeks. (Months? Years...?)

The Arduino and the isolating power supply can be fitted inside the Porter-10's case. The sensor goes outside:



Note carefully how the current-sensor is threaded: exactly two passes of the main battery wires should thread through the centre of the transducer. Here, we've threaded the positive battery wire one way, and the negative battery sire the opposite way. Direction of threading does matter:



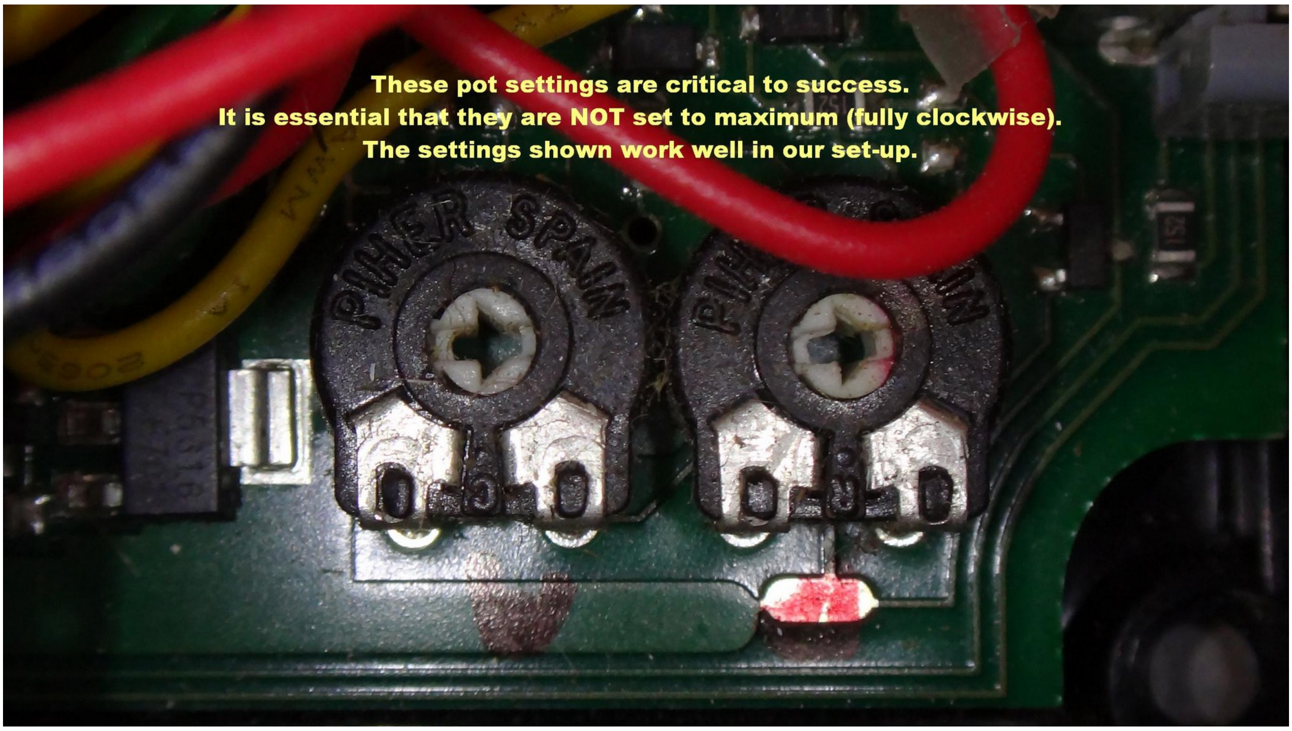
The pictured threading method has significant disadvantages due to physical movement and stray currents induced from nearby wiring. Future builds will instead thread the sensor using rigid copper wire wrapped twice through the centre and locked securely in place with adhesive, with nearby power cables also locked into position relative to the sensor. This should ensure more consistent readings. (Bearing in mind that *consistency* is more important that absolute *accuracy*.)

Setting the Porter-10

It is **vital** that the Porter-10 potentiometers are set up correctly. As far as I can figure, the potentiometers control how rapidly the output PWM is allowed to ramp up and ramp down - i.e. they effectively limit the slope of the output voltage/time curve. So - unlike many controllers - the "decel" pot does not directly control the maximum limit on regen. Instead it controls how rapidly the PWM duty-cycle is allowed to 'fade away' to zero when you let go the throttle.

This makes the controller a bit tricky to set up in this application. Too slow a response means the PID gets slow feedback to its inputs, causing it to oscillate. And even if you conquer the oscillations, you still never get any useful regen. So you'd think that turning the pots all the way to the right - to force the Porter output to immediately follow the demanded input - would be the thing to do. It isn't. Seriously, beware that if the pots are turned all the way to the right to maximise the response, things go properly haywire. Just Don't Try It, OK? Or, if you do try it, make sure your test motor is securely bolted down and free to rotate. And you have lots of spare fuses. And you have a flyback diode wired-in across the fuse-holder. And an easily-reached Big Red Off Switch.

The best setting seems to be about like this:



Threading Notes

The current sensor is threaded for +/-125A. This might seem an excessive range, given that we're trying to control only 0A to 30A.

However, the core controller is a PWM on/off device, a typical LiPo power supply has close to zero internal resistance, and the load motor (at standstill) has an extremely low impedance. So at motor start-up - even when using very low duty-cycles - the *peak* current of the PWM cycle can be many times the demanded *average* current. We cannot afford to allow the sensor to crop the signal (because we would underestimate the actual current) so we need to thread it for very large currents.

Core Controller Notes

We're sensing *battery* current, not motor current, because we want to protect the battery fuse. Motor average current can be much higher than battery average current, something to keep in mind when choosing the core controller's maximum current capability (which is rated on motor current.)

Software

The software is written for Arduino using the standard Arduino IDE v1.8.3 on Linux. (We have seen issues with some older Arduino IDE versions - especially on Windows - which may require forward declarations to be added to some code. We haven't tested this.)

Note that our code has multiple issues in design, implementation, coding misconceptions, and indeed some complete c*ck-ups not worthy of a coding professional <embarrassed emoji>. We've annotated some of these and we do know about the others, so please don't tell us about them. Our playtimes are busy and brief and this code does work as-is to achieve its main goal. Which is to drive the motor reasonably hard but without blowing fuses. Fixing the errors - without enhancing the code in other ways - changes the software behaviour in undesirable ways. I.e. it blows fuses and/or reduces the motor power. So, please begin by setting the IDE's warning-level to “none”, then build it, flash it, and forget it. You can hack it at your leisure later :-)

One improvement, that you may like to add, is to wrap the grabbing of <setpoint> and <input> values in the main loop as an ATOMIC operation. We have done so in the devel version (not yet uploaded). Otherwise there is risk that these values will be occasionally corrupted.

Find the stable code at:
https://github.com/MechanicalCat/HackyRacerPID

The code uses our own internal DEBUG library. So either strip out our debugging from the code, or get the library from:
https://github.com/MechanicalCat/Debug
This library is not 'properly' packaged. But it's so small you'll easily figure it out. Just copy the files Debug.h and Debug.cpp into a directory at .../libraries/Debug

You'll also need the third-party Arduino PID library. See:
https://playground.arduino.cc/Code/PIDLibrary

We're working on a new bug-fixed version with a more elegant method of managing fuse heating. We're hoping this will work on cheaper (£10) dumb controllers, rather than only on the more expensive intelligent Porter-10. So feel free to contact us via www.MechanicalCat.org if you'd like to ask if there's an updated version of our code available.

Software Notes

Instance <sensor> of class <SenseI> maintains a rolling-average of the sensed battery current from 1000+ samples spaced over several milliseconds. Clearly this is slightly laggy and slightly imprecise - but it has to be because it is necessarily averaged over a/some duty cycle(s). For the purposes of this software this rolling-averaged current is deemed to be the *instantaneous* current.

Instance <fuse> of class <Fuse> maintains a rolling average of the instantaneous current over the past 3 seconds and the past 90 seconds. It also maintains a copy of the immediate instantaneous current.

Class <Fuse> allows comparison of the various average/instantaneous currents against the fuse's documented/guessed performance, and can suggest a maximum safe current that the fuse is capable of bearing, based on the recent history of current-flow (and therefore heating) that the fuse has endured. It does this by applying cool-down periods after certain current limits are exceeded.

Instance <motor> of class <Motor> is the interface to the core open-loop motor controller (the Porter-10). The method <drive> sets the controller's PWM duty cycle.

Instance <thumb> of class <ThumbLever> reads the user's input as a user-demanded current.

In the main loop, instance <pid> of class <PID> constantly modifies the Porter-10's output (via <motor>) in order to maintain <input>* (read from <sensor>) at the same value as <setpoint> (read from <thumb>). Meanwhile <fuse> constantly adjusts the maximum permitted value of <setpoint> to prevent the physical circuit fuse being blown. In high-load usage, <input> should more-or-less track <setpoint>. But, when the physical motor is running at high speed and low load, the value of <input> may be lower than <setpoint>, even though the duty cycle is pushed all the way to 100%.

Also in the main loop, a separate check is kept on <thumb>: if the user's demand is ever zero (i.e. they let go the thumb lever) then all the above logic is short-circuited and the motor is immediately set to zero power. There are three reasons for this kludge:

- The 'off' response is as immediate as possible, although there may still be lag in the Porter-10 itself.
- The Porter's full regen/braking capability is engaged.
- The need to precisely calibrate** IZERO is obviated.

*The variable <input> is not as confusingly-named as it seems. It means the input (i.e. the feedback) that the *PID* receives from the system.

***Of all the reasons, this is - surprisingly - the most significant.*

Enhancements

The software records the sum of charge used from the battery since boot. Clearly this could be useful information to display to the driver, along with related data such as instantaneous current and peak current. If the charge-used variable were made persistent across restarts (either by storing in flash or by permanently powering the Arduino) then it could be further used to enforce battery depletion limits. Depending, naturally, on how [in]accurate it is.

Beware, though, that the MurataPS DC-DC converter has a power limit of only 3W, and the Arduino 5V supply will also have a limit. So adding, say, an OLED display could exceed these limits and destroy a component.