Aristos Athens

**Catheter Hardware Update Notes**

**Overall Plan:**

-To move to a motor to a specified position, we will write a PWM signal to the motor, then read the encoder until we see the motor is at the correct position. Then stop the PWM. In this simple set up we aren’t controlling speeds or torques, only position.

-I think the biggest challenge will be reading the encoder signals. Using interrupts on Arduinos can be a bit of a pain.

**Microcontroller:**

Moving Catheter setup to ArduinoMEGA (processor: ATMega2560). The Arduino should be sufficient but note that the ATMega2560 is still 8-bit. The main issue is precision (no double or long data types available natively. ie double and float are the same). If we are using brushed motors, no new hardware is needed. The ATMega2560 has 54 I/O pins and 15 of them can be used for PWM. Main issue is that it only has 6 interrupt pins natively. This can potentially be extended to 18. See Reading Encoders section.

**PC to microcontroller communication:**

Arguably the biggest challenge. Since Python is interpreted by default, you would need to load your code plus the entire python language onto the processor. No way an Arduino has enough memory for that.

We can run the script on the desktop which then communicates in real time to the Arduino. Probably our best bet here. PySerial is a package (API?) that lets the Arduino send info back and forth between a PC using a USB cable. There are a number of tutorials for this:

<https://playground.arduino.cc/Interfacing/Python>

<http://www.instructables.com/id/Arduino-Python-Communication-via-USB/>

^We can probably skip all of that and go directly to PythonBridge, which uses PySerial and creates a nice environment which lets us edit Python in our own IDE then simply send the files to the Arduino with their interface.

<https://github.com/willbelr/bridge>

**Hardware:**

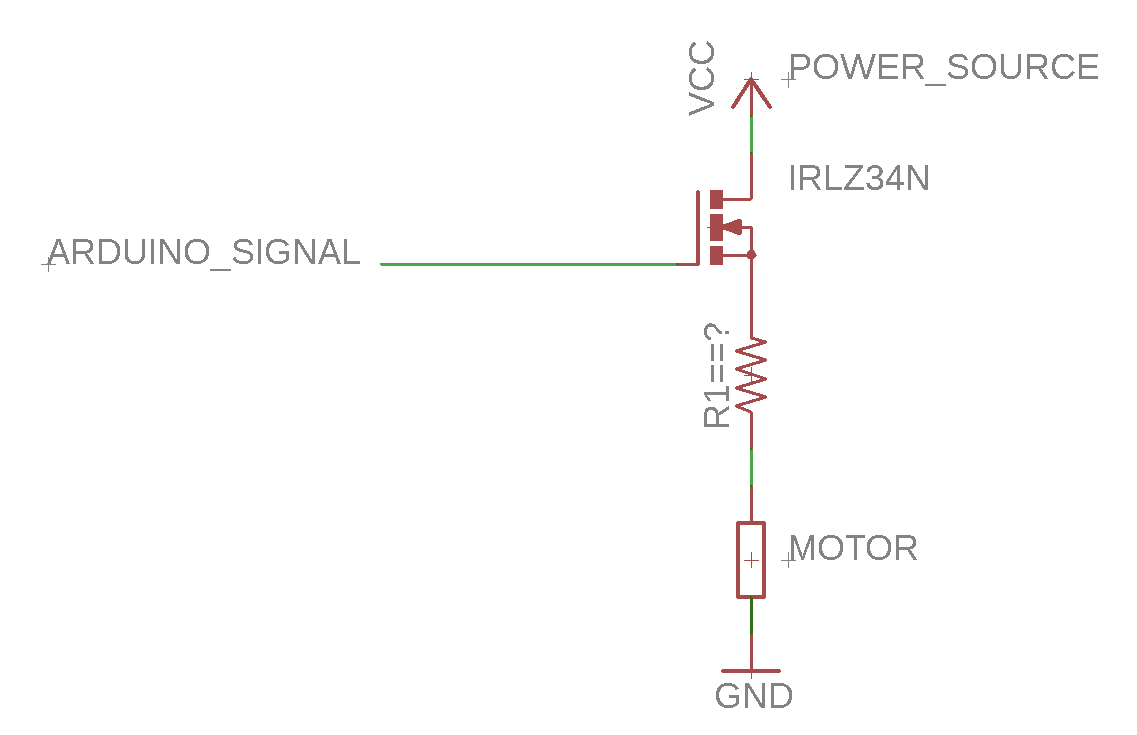
-The ArduinoMega 2560 has 15 PWM pins, so we don’t need any extra boards.

-If the motors are brushless we will need ESCs (electronic speed controllers) between the microcontroller and each motor. These are quite easy to control. Simply send a signal of a specified length to the ESC, and it does the rest. As an example, the “oneshot125” protocol is as follows: To turn off the motor, send a high signal for 125 microseconds. For full speed, 250 microseconds. For half speed, 187.5 microseconds. There is also oneshot250 and “PWM” (oneshot1000).

-Use the existing power supply for the motors, Arduino pins can’t supply enough current. Use PWM pins to connect to the gates of Power N-type Mosfets to switch the power.

-Use Arduino to power encoders. It’s extremely important that encoder Ground matches Arduino Ground.

-All we need is: **10 power transistors + new wiring**. We will simply need to create some wires which connect the Arduino to the gate of each transistor, then connect each motor to the source of each transistor. We will also need wires to connect the encoders to the Arduino.



A few things to note:

- The simple plugs to the motors are the power wires, not the signal wires. The signal wires are wrapped up in the black cables which run back to the pizza box boards. We will need to unplug these from the pizza boxes and connect to the Arduino pins.

-Vcc is the power supply. I don’t know what voltage it gives.

-I’m not sure if R1 is necessary, and if it is, what value to use. We just want it there to limit current. If we do use it, the value should be small (<<<1Kohms).

**Transistors**

-The transistor needs to be able to handle the necessary motor currents, so we should use power transistors to be safe. The IRLZ34N should be fine for this. It can handle 20-30 amps. (<http://www.irf.com/product-info/datasheets/data/irlz34n.pdf>)

-N type. Requires VGS = 2V to turn on. This type of transistor compares Gate voltage (Arduino pin) to Source voltage (Ground). Therefore it is important that Arduino Ground matches power supply Ground. This makes me concerned you can get a “ground loop” where Arduino Ground does not match Ground of the power supply, and you continuously leak current. Ask Hossein about this.

-We need to compare the transistor’s switching speed to the Arduino’s PWM frequency. For a single wave the transistor has latency of about 160 nanoseconds. We should aim for a PWM frequency at least 10 times larger than this. So PWM frequency should be less than 100kHz or so. I think default is 31kHz, so shouldn’t have to change anything here.

-RDS = 0.06 Ohms when the transistor is on. For the whole circuit (one motor) RTotal = RDS + R1 + Rmotor. Need to add enough resistance so that neither the transistors or motors burn out. R1 *might* not be necessary.

Wiring:

Gate : Arduino Output Pin

Drain (+) : Power Source

Source (-) : Resistor/Motor

**Encoders**

-From what I can tell, the motor encoders are these: <https://www.usdigital.com/products/e4p>.

The encoders are 6 pin, instead of 4 pin. So connect them as follows:

G : Ground

A : Arduino Digital Input Pin

A- : Ground

+ : 5V

B : Arduino Digital Input Pin

B- : Ground

-These work by using two PWM signals. The difference in phase determines the direction of rotation (see link). The phase difference represents motor *speed*, so to get position we will have to integrate this value.I talked to a rep from US Digital, and he provided this info:

Hi Aristos,

For the quadrature output of the E4P encoder, the phase difference of 90 degrees refers to ¼ of the quadrature cycle (90 electrical degrees, rather than 90 mechanical degrees).

You can find more information about the quadrature cycle on our website here:

<https://www.usdigital.com/support/glossary#glossary_quadrature>

Or by looking at the timing characteristics of our EM1 module here:

<https://www.usdigital.com/products/encoders/incremental/modules/EM1>

The guy is named Joshua Morris ([support@usdigital.com](mailto:support@usdigital.com)) and he is super helpful, but I think he got sick of me by the end.

**Reading Encoders**

-Two options: using PulseIn() function or using interrupts. Interrupts are better.

-Arduino provides the PulseIn() function to read pulse lengths. The problem with this is that it starts reading from the time the function is called, which does not necessarily match up with the beginning of the pulse. So we might have to call this function sequentially, 20 times (2 for each encoder). It could be something like this:

While (pin == high) { do nothing }

long Pulse\_length = PulseIn(pin, HIGH);

This will work, but it’s kind of jank, because we aren’t measuring all motors simultaneously, we are measuring them sequentially. Also, PulseIn() is blocking code…

<https://www.arduino.cc/reference/en/language/functions/advanced-io/pulsein/>

-The real way to do this is with interrupts. To whomever is reading this: If you know what interrupts are, skip to the next paragraph. If not I’ll briefly explain them. An interrupt is a way to monitor pins outside of software. For example, you might want to constantly check if a button is pressed, so your main loop calls checkButton() to check Pin 1. An interrupt is a faster and cleaner way to do this. If the voltage at Pin 1 changes, the actual hardware on the microcontroller will literally interrupt, or pause, your code (called the “foreground” code) and run a piece of code that is stored elsewhere (called an “ISR” – Interrupt Service Routine). Once the ISR is done, your foreground code continues as though nothing happened. In Arduino and most compilers you write your ISRs in the same place (same file) as your foreground code, but must mark them as ISRs. ISRs should generally be as short as possible. You do not pass parameters to an ISR, and they cannot return values. Instead, you should create global variables and store ISR info there.

Interrupts are particularly useful for timing-critical operations. For example, we want to read the frequency of an incoming signal. If we simply had a checker that ran in our normal loop, it’s possible we would miss rising or falling edges, and our time measurements would be wrong or unreliable. Instead, set up an interrupt that triggers every time the pin changes state. Then in the ISR you will simply read the time and store this value. Then use this information outside of the ISR.

Here is a bit more info on interrupts:

<https://www.allaboutcircuits.com/technical-articles/using-interrupts-on-arduino/>

<https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/>

-There are only 6 digital pins that can handle interrupts on the Arduino Mega (AtMeg2560). There is a library which extends this functionality to 18 pins. I’ve looked into it, and it seems fast, but we should note that there is a software portion that makes this work, so it won’t be as fast as native interrupts.

<https://github.com/GreyGnome/EnableInterrupt/wiki/Usage>

**Challenges:**

1. Actually getting the Python-->USB-->Arduino connection working. (Jake has done this before) The packages I linked above should do this for us, but weird bugs seem inevitable. Additionally, we need to figure out the arduino’s usb address. The tutorials give us scripts for finding it.
2. Knowing what PWM values to send. As is we just send AurisLowLevel a “motor speed” and it converts that to the appropriate PWM duty cycles and direction. If we don’t care about getting the speed exactly right then this becomes pretty easy. A duty cycle must be between 0 and 100, so we just scale our “motor speed” from 0 and max\_speed to 0 and 100.
3. Reading the encoders. We need 20 interrupt pins, but the Mega only has 6. With the library we can get 18, still 2 short.