PID Control:
A brief introduction and guide, using Arduino.

(SWR 26 Sep 2011)

Overview: PID (Proportional, Integral, Derivative) control is a widely-used method to achieve and maintain a process set point. The *process* itself can vary widely, ranging from temperature control in thousand gallon vats of tomato soup to speed control in miniature electric motors to position control of an inkjet printer head, and on and on. While the applications vary widely, the approach in each case remains quite similar. The PID control equation may be expressed in various ways, but a general formulation is:

$$\text{Drive} = kP*\text{Error} + kI*\Sigma\ \text{Error} + kD * dP/dT$$

where Error is the difference between the current value of the process variable (temperature, speed, position) and the desired set point, usually written as Error = (Value-SetPoint); ΣError is the summation of previous Error values; and dP/dT is the time rate of change of the process variable being controlled, or of the error itself. The proportional coefficient kP, the integral coefficient kI, and the derivative coefficient kD are *gain* coefficients which *tune* the PID equation to the particular process being controlled. Drive is the total control effort (often a voltage or current) applied to actuators (heater, motor, valve) to achieve and hold the set point.

Tuning methods: Not all of the terms in the PID equation are necessarily used. Fitting the PID approach to a particular control problem involves *tuning*; deciding which terms to include, and determining what the gains should be for those terms. The Wikipedia page http://en.wikipedia.org/wiki/PID_controller presents the basic PID approach and outlines some tuning methods. Although there are analytical approaches, rules of thumb, and specialized software among other methods for selecting gain coefficients, the gains are often arrived at, particularly in small model systems, by observing the actual response of the process to a particular set of coefficients and adjusting them until good control is achieved.

Coding a PID control algorithm: Code for a PID system can be rather simple. The following is an example of some *pseudocode* to do PID:

```
PID:
     Error = Setpoint - Actual
     Integral = Integral + (Error*dt)
     Derivative = (Error - Previous_error)/dt
     Drive = (Error*kP) + (Integral*kI) + (Derivative*kD)
     Previous_error = Error
     wait(dt)
GOTO PID
```

This pseudocode is not written in any particular computer language or for any particular microcontroller, but does lay out the basic logical steps to achieve control using the PID approach. It also leaves out a few details which usually need to be included depending on the particular control problem and the particular controller being used.

The program code shown below is written in the C-like Arduino language which you can use as a guide to developing your own PID function. You must define the variables and pin numbers for the position potentiometer (Position), motor drive pin (Motor), and motor direction pin (Direction).

```
//-------------------- Calculates the PID drive value  --------------------
      Actual = analogRead(Position);
      Error = SetPt - Actual;

      if (abs(Error) < IntThresh){          // prevent integral 'windup'
         Integral = Integral + Error;       // accumulate the error integral
      }
      else {
         Integral=0;                          // zero it if out of bounds
      }
      P = Error*kP;                         // calc proportional term
      I = Integral*kI;                      // integral term
      D = (Last-Actual)*kD;                 // derivative term
      Drive = P + I + D;                    // Total drive = P+I+D
      Drive = Drive*ScaleFactor;            // scale Drive to be in the range 0-255
      if (Drive < 0){                       // Check which direction to go.
         digitalWrite (Direction,LOW);      // change direction as needed
      }
      else {                                //   depending on the sign of Error
         digitalWrite (Direction,HIGH);
      }
      if (abs(Drive)>255) {
         Drive=255;
      }
      analogWrite (Motor,Drive);            // send PWM command to motor board
      Last = Actual;                        // save current value for next time
}
```

PWM value: The value for the PWM duty cycle parameter in the **analogWrite**() instruction above must be an integer in the range 0-255. So, as you calculate the sum of the P,I, and D terms you need to scale the final Drive value to fit into the 0-255 range.

Integral term: Use of the integral term can be problematic. In PID control systems there is a problem referred to as “windup”, which occurs as follows. The integral term sums the error history over time. If a system starts far from the final desired set point, the initial errors will be large, and the integral term will quickly grow very large. This accumulated integral usually produces a dominating effect which prevents the system from quickly achieving the set point. To avoid this problem, a number of methods have been employed. The scheme shown in the code above is to “zero out” the integral term unless the error is sufficiently small. Specifically, the test here is to check if the current error is less than some test value. In the code above the test value is called IntThresh. Including this test allows the integral term to operate only after the system has approached close to the final set point. The integral term then acts to remove any small residual error so that the system may converge to the final set point.

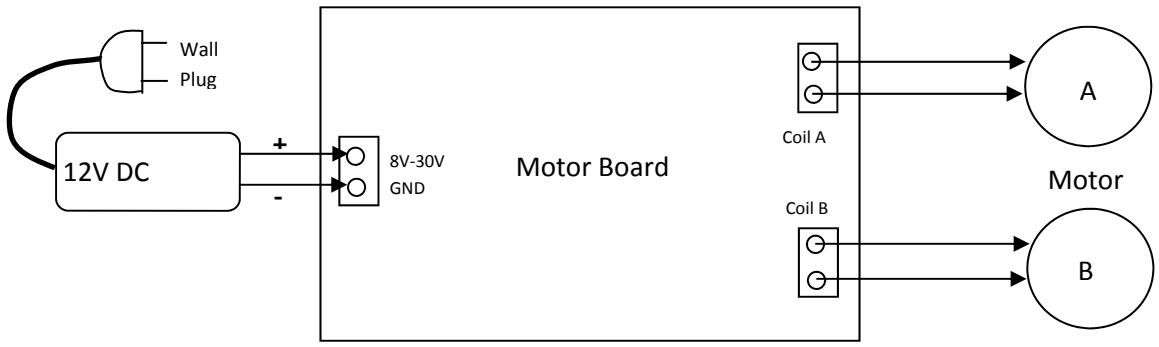
Tuning: As mentioned earlier, the process of determining appropriate values for the gain coefficients kP, kI, and kD refers to "tuning" the system. A simple empirical approach is to start by zeroing the integral and derivative gains, and just use the proportional term. Setting the proportional gain increasingly higher will finally cause the system to oscillate. This is bad behavior; don't allow it to continue. Reduce the proportional gain until you are just below the point of incipient oscillation. You can then try bringing up the derivative gain, which should act to forestall the start of oscillatory behavior. And finally adding a small amount of integral gain may help bring the system to the final set point.

The best coefficients for a given control system depend on the goals of the system. Bringing a subway train smoothly up to speed with no oscillations or overshoot is one goal; rapidly achieving a set point where some overshoot and oscillations are an acceptable tradeoff for fast response is a different goal. Different control goals require different tunings.

Motor Driver Usage

Connections: The motor board plugs directly onto the Arduino stacking connectors. You will attach your motor(s) to the two screw terminals labeled "Coil A" and "Coil B" on the motor board. You will also connect power for the motors at the screw terminal labeled "8V-30V" and "GND". This motor-power connection **only** powers the motor board, and is separate from the Arduino board power supply. The Arduino board itself can be powered either by a USB connection to a computer, or by a separate DC power source plugged into the circular DC power connewctor on the Arduino.

Connections to the motor board are shown in the diagram below:



When making your connections BE SURE to get the correct polarity on the 12V DC connector. If you reverse the connection the motors will not run, and you may possibly burn something on the motor board.

The four outputs can each source a maximum current of about 2 amps max, and for continuous duty at the full current a heat sink and/or cooling fan would be required to prevent components from overheating. However, in our application the full current will typically only be used for short periods as the control system drives the motor toward the set point, so overheating shouldn't be a problem. Once near the set point (and this should take only a few seconds) the motor current will drop to low values. Assuming of course that there are no bugs in your software.

The instruction sequence for sending a command to the motor driver is:

```
digitalWrite(dirpin,value);   // set direction
analogWrite(motor,drive);     // send PWM command
```

The default pin Arduino assignments for the motor driver are:

Pin 3: PWM for motor A
Pin 12: direction for motor A
Pin 11: PWM for motor B
Pin 13 : direction for motor B

An example command would be:

```
digitalWrite(12,LOW);          // set direction motor A
analogWrite(3,117);            // send PWM command motor A
```

Note that 0 or 1 can be substituted for LOW or HIGH, respectively, in the direction instruction. Also, the PWM parameter must be 0-255, which maps to a percent duty cycle of 0-100%. PWM values outside the 0-255 range will produce incorrect behavior. In the example above the PWM value of 117 corresponds to a duty cycle of 117/255 = 46%. In addition, program readability can often be improved by using names for the pin numbers with the **`#define`** construct near the top of your program, i.e., **`#define`** `PWMA = 12`, after which you could write **`digitalWrite`**`(PWMA,LOW).` Or even **`digitalWrite`**`(PWMA,0).`

Note also that in the Arduino compiler the PWM frequency cannot be readily changed from the default value of 490 Hz. To change the PWM frequency the internal TCCRxB register must be modified. The "x" refers to the particular internal timer (0, 1, or 2) controlling the PWM pin you are using. Refer to the Barrett book for details. For our use with motors the 490 Hz works OK, but for some other applications other frequencies may work better. Although the Arduino Uno has 6 PWM channels, the internal timer 0 used for the PWM channels on pins 5 and 6 is also used for the **`milli()`** and other timing functions. If you change the frequency on timer 0 from the default 976.56 Hz these timing functions will give incorrect results.

Data Logging

Optimizing the performance of your system requires that you capture performance data for analysis so that you can quantify effects of hardware and software changes. These will be the data with which you justify any design changes in the quest for system optimization and improved performance.

To capture the motor-drive performance information your program can output data over the USB port using the **Serial.print** instruction. You can include this instruction in your control loop to observe intermediate control values, such as the current potentiometer voltage, as your system proceeds toward achieving a set point. This will give you a time course of position which you can then analyse for velocity and acceleration behavior to quantify the motor control dynamics.

The serial output data can be captured using the built-in serial terminal facility of the Arduino development environment. Under the Tools pulldown menu select Serial Monitor. When the serial monitor opens, in the lower right-hand corner select the correct ***baud rate*** (the communications transfer rate) to agree with the rate you specified in the **`Serial.begin`**`(baudrate)` instruction in your program. Selecting a baud rate of 115200 will give you quick data transfers. When your data run is complete you can copy/paste the data displayed in the Serial Monitor window into an Excel spreadsheet or other program for graphing and analysis.

Here's an outline of how to set up the serial data transfer:

```
float value1 = 1.234, value2 = 2.345, value3=3.456; // set up some test values
float value4 = 4.567, value5 = 5.678;

void setup() {
   Serial.begin(115200);                              // set the communications rate
}

void loop() {
   Serial.print(millis()); Serial.print(",");         // print time and a comma
   Serial.print(value1,4); Serial.print(",");        // print value and a comma
   Serial.print(value2,4); Serial.print(",");
   Serial.print(value3,4); Serial.print(",");
   Serial.print(value4,4); Serial.print(",");
   Serial.println(value5,4);                          // final value and newline
}
```

The first value printed (**`millis`**() function) will be the number of milliseconds since the Arduino was last reset. This will give you a time base for your data. The optional "4" in the **`Serial.print`** instruction specifies how many digits to the right of the decimal to print. Notice that the final "value5" is printed using the **`println`** version of the **`Serial.print`** instruction. This appends a final character (called "carriage return" or "newline") to the end of your series of output values such that a series of output rows will be produced, each row

containing the time stamp and the 5 data points. Each value in a row will be delimited by a comma to format the data for importing into Excel or other analysis program.

PWM Frequency Control – Arduino Uno

There are 6 PWM channels available. The instruction to produce PWM output is **analogWrite(pin,Duty)**, where **pin** must be 5,6,9,10,11,or 3, and **Duty** is the duty cycle, entered as 0-255 corresponding to 0-100%. The default PWM frequency is 490 Hz.

To change the frequency an additional instruction is required. The PWM frequency is controlled by three internal timers: timer0, timer1, and timer2. The instruction to change the PWM frequency is **TCCRnB = (TCCRnB & 0xF8) | i** where i is chosen depending on which timer is associated with the PWM pin whose frequency you want to change. Note that the PWM pins are associated in pairs with the same timer. Thus if you change the PWM frequency for pin 9, you will get the same frequency for PWM on pin 10. However the duty cycles can be different on the two pins. The table below summarizes the options available.

Timer0 (TCCR0B for pins 5 and 6)

i	freq
1	62500 Hz
2	7812
3	976
4	244
5	61

Timer1 (TCCR1B for pins 9 and 10)

i	freq
1	31250 Hz
2	3906
3	488
4	122
5	30

Timer2 (TCCR2B for pins 3 and 11)

i	freq
1	31250
2	3906
3	980
4	490
5	245
6	122
7	30

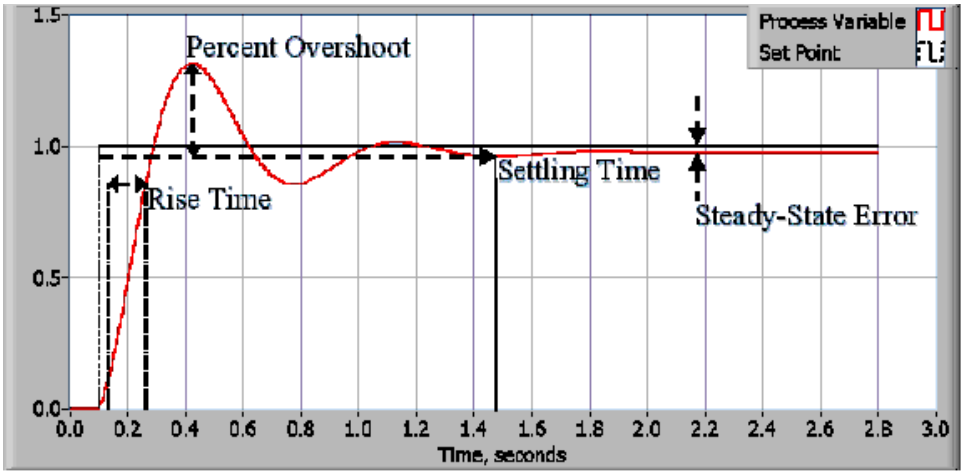
Example frequency change instruction: **`TCCR2B = (TCCR2B & 0xF8) | 2`**
(sets PWM pins 3 and 11 for 3906 Hz)

Example program:

```
void setup()
{
  pinMode(3,OUTPUT);              //make pin 3 an output
  TCCR2B=(TCCR2B&0xF8) | 2;       //set PWM frequency to 3906 Hz for pin 3 (and 11)
}
void loop()
{
  analogWrite(3,127);             //do 50% PWM on pin 3 at the frequency set in TCCR2B
}
```

In general you should avoid changing the PWM frequency on pins 5 and 6 since they use timer0, which controls the **`delay`** and **`milli`** functions. These functions will return incorrect results if the frequency of timer0 is changed from the default. The factor 0xF8 is a mask so that you only affect the bits for the frequency in the TCCRnB register when OR-ing with the | operator.

Below is a "textbook" example of a control system response to a step change in the set point, showing some typical ways of characterizing the response.



Here we would say the steady-state response is achieved in about 1.5 seconds with an initial rise time of around 150 milliseconds and a 30% initial overshoot. If objectionable, the remaining steady-state error might be "cured" by adding some integral action

Some references

Arduino language reference: http://www.arduino.cc/en/Reference/HomePage

Barrett, Steven F. 2010. Arduino Microcontroller: Processing for Everyone. Morgan & Claypool, 325 pp. ISBN: 9781608454389. Available online at UCSD library.

Margolis, Michael. 2011. Arduino Cookbook. O'Reilly. 631 pp. ISBN: 9780596802479.

Prinz, Peter and Tony Crawford. 2006. C In A Nutshell. O'Reilly. 600 pp.
ISBN: 9780596006976