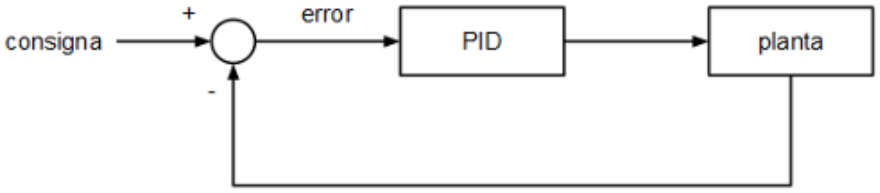
Control de velocidad tipo PID para un motor DC

miércoles, 21 de enero de 2015, 23:25 - **Desarrollo Arduino**, **Desarrollo embebido**

Uno de los controladores más utilizados es el tipo PID (**Proporcional Integral Derivativo**). A lo largo de este post se abordará la implementación de uno en Arduino para controlar la velocidad de un motor DC.

Un poco de teoría

Cuando se quiere controlar una planta (en nuestro caso un motor DC), lo más habitual es plantear un lazo de control estándar:



La señal que entra al controlador es la medida que queremos que alcance la planta (llamada "consigna" en teoría del control) menos la medida de salida de la planta o, lo que es lo mismo, el error. El objetivo del controlador será siempre minimizar el valor absoluto del error (que tienda a cero) actuando sobre la entrada de la planta.

Para profundizar bien en el estudio del control habría que ver las transformadas de Laplace, los polos y los ceros del sistema y, para el caso discreto, lo ideal sería un estudio basado en la transformada Z estudiando también la ubicación de los polos y los ceros. Sin embargo me centraré en el estudio y la implementación de un controlador estándar: el PID.

PID

Los controladores PID son un tipo especial de controlador que combinan la acción proporcional (P), la acción integral (I) y la acción derivativa (D) sobre el error. Si a la entrada del controlador (el error) la llamamos e(t) y a la salida del controlador (la entrada a la planta, en nuestro caso la entrada al motor DC) la llamamos u(t). Podemos definir un PID de la siguiente manera:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

Como se puede apreciar, la acción proporcional vendrá determinada por la constante K_p, la acción integral por la constante K_i y la acción derivativa por la constante K_d.

1. La acción proporcional K_p hace que el error en estado estacionario tienda a cero.

2. La acción integral K_i, al ir sumando los errores en el tiempo (integral), tiende a eliminar el error estacionario generado por la acción proporcional.

3. La acción derivativa K_d tiende a suavizar las variaciones en el error.

Para determinar los mejores valores de cada una de las constantes, lo ideal es realizar un estudio mediante la transformada de Laplace y buscar la mejor ubicación de los polos y los ceros del controlador PID para que se obtenga el comportamiento deseado.

En este caso se ha optado por realizar pruebas empíricas con valores bajos e ir probando diferentes combinaciones.

Implementación a nivel hardware

En este caso la planta es un motor DC del que vamos a controlar su velocidad mediante la salida PWM de 8 bits (0 a 255) y 5 voltios. La salida PWM la conectamos a la base de un transistor NPN de potencia (en este caso un BD139) montado en configuración de emisor común.

Enlaces

Principal
Contacta Conmigo
Estadísticas

MSX
Call MSX
Síntesis Analógica Musical
Chameleon
Música
Soft
Gameboy Advance
PIC
Nintendo DS
Guineo

Matrixsynth
aOrante Blog
A través de las gafas de una ingeniera
Curiosidades de la programación

Entrar

Redes rociales

Seguir a @avelinohm

Calendario

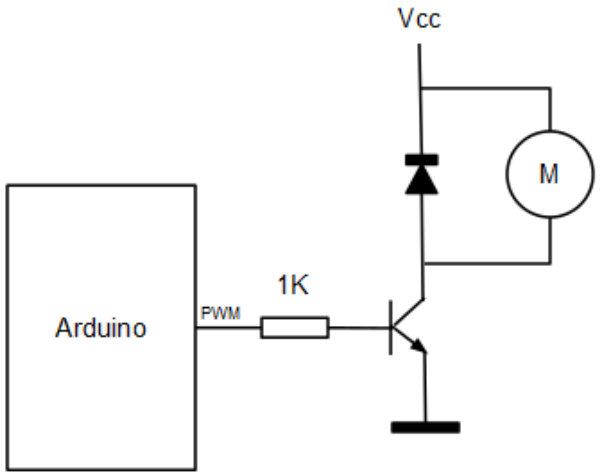
«	Febrero 2016					»
Dom	Lun	Mar	Mié	Jue	Vie	Sáb
	1	2	3	4	**5**	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29					

16/03/16

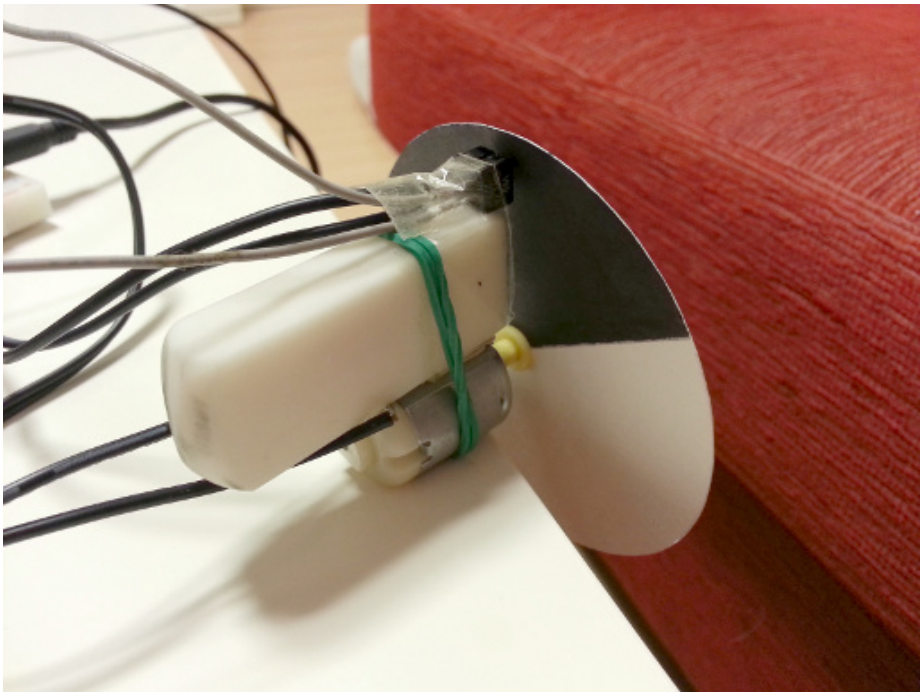
Agüita con los montunos
Vaya regalazo
Ya tengo el número 7 de la Call MSX
Memoria compartida en C++
Vúmetro LCD

Archivo

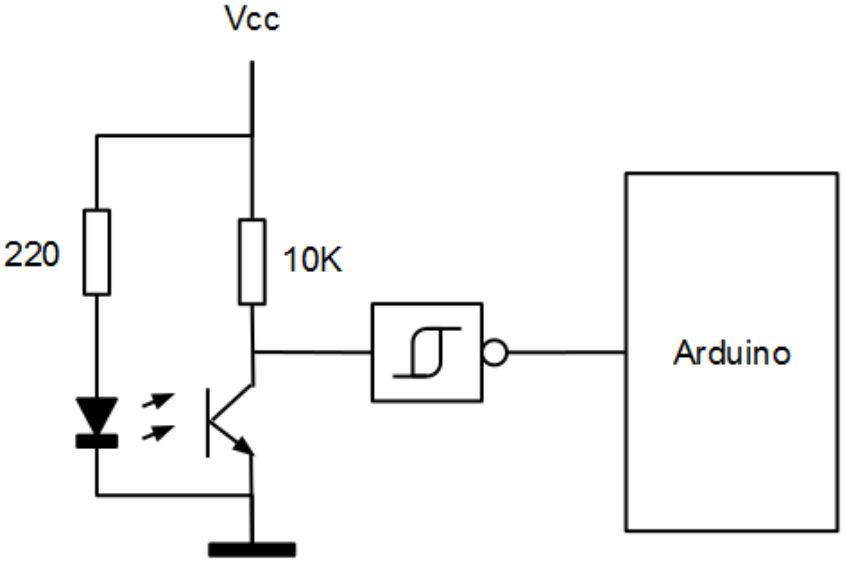
Ver Archivos
2016
febrero
Programación del microcontrolador LPC810 en C++ desde cero
05/02/16
enero
2015
diciembre
octubre
septiembre
julio
mayo
abril
febrero
enero
2014
diciembre
noviembre
septiembre
agosto
junio
mayo



La lectura de la velocidad angular la hacemos utilizando un disco pintado (mitad blanco y mitad negro) conectado al eje de rotación (para que gire) y un sensor reflexivo de infrarrojos CNY70 (del que se utiliza en los robots sigue líneas).



Polarizando el fototransistor y el led infrarrojo y acondicionando la señal con una puerta inversora de tipo schmitt, ya tenemos un flanco de subida o de bajada por cada vuelta que da el disco.



Implementación a nivel software

Lectura de la velocidad

Para obtener la velocidad de rotación lo más eficiente es conectar la salida del inversor schmitt a una entrada del microcontrolador que permita disparar interrupciones internas en cada flanco de bajada o en cada flanco de subida. El pseudocódigo sería como sigue:

```
rpm = 0
anterior_t = 0

cada vez que haya un flanco de subida hacer:
    t = microsegundos
    incremento = t - anterior_t
    rpm = (1 / incremento) * 60000000
    anterior_t = t
fin interrupción
```

De esta forma tenemos los rpm a los que va el motor. Nótese que esta implementación no detecta la velocidad de 0 rpm. Para detectar la velocidad de 0 rpm habría que incluir un timer que, pasado un tiempo determinado, si no se produce la interrupción, asuma que el disco se ha parado (rpm = 0). En este caso no se ha implementado esta funcionalidad por simplicidad.

Implementación del PID

Para implementar el controlador PID en el Arduino (o en cualquier otro microcontrolador) tenemos que discretizar la ecuación diferencial que relaciona u(t) con e(t). Separamos primero dicha ecuación diferencial en partes:

$$u(t) = u_p(t) + u_i(t) + u_d(t)$$

Siendo:

$$u_p(t) = K_p e(t)$$

$$u_i(t) = K_i \int_0^t e(\tau) d\tau$$

$$u_d(t) = K_d \frac{de(t)}{dt}$$

La discretización de $u_p(t)$ es trivial:

$$u_p[k] = K_p e[k]$$

La discretización de $u_i(t)$ asumiento un período de muestreo de T lo suficientemente bajo la podemos calcular aproximando la integral mediante una suma de áreas de rectángulos de base T y altura $e[k]$:

$$u_i[k] = K_i \sum_{n=0}^{k} T e[n] = K_i T \sum_{n=0}^{k} e[n]$$

De la misma manera, la discretización de $u_d(t)$ asumiento un período de muestreo T lo suficientemente bajo la podemos calcular aproximando la derivada mediante el cálculo de la pendiente de la recta que une $e[k-1]$ con $e[k]$:

$$u_d[k] = K_d \frac{e[k] - e[k-1]}{T}$$

El PID discretizado nos quedaría, por tanto, de la siguiente manera:

$$u[k] = K_p e[k] + K_i T \sum_{n=0}^{k} e[n] + K_d \frac{e[k] - e[k-1]}{T}$$

Esta ecuación en diferencias finitas sí es fácilmente implementable en cualquier sistema. En el caso de Arduino podríamos realizar la siguiente implementación:

```
struct pid_controller {
  float kp, ki, kd;
  float delta;
  float sum;
  float prevError;
};

void pid_controller_init(struct pid_controller &pid, float delta, float kp, float ki, float
  pid.delta = delta;
```

Categorías

General
Música
  Desarrollo DSP
    Hardware para música
    Soundart Chameleon
    Roland Fantom XR
  Software para música
  Hardware para música
MSX
  Desarrollo para MSX
    SDCC
    Sonido para MSX
Desarrollo en general
  Desarrollo Android
  Desarrollo Arduino
  Desarrollo embebido
  Desarrollo Teensy
  Desarrollo LPC810
  Desarrollo FPGAs y CPLDs
Literatura
Consolas
  Atari 2600
  Gameboy Advance
  XBox
  Nintendo DS
PIC

Búsqueda

Búsqueda
Enviar

Últimos Artículos

Programación del microcontrolador LPC810 en C++ desde cero
Salida de audio de alta calidad con la placa Teensy
Luces de Navidad controladas por FPGA
Implementación de un receptor serie asíncrono sobre FPGA
Implementación del algoritmo de multiplicación de Booth en VHDL sobre una FPGA
Síntesis musical mediante modelado analógico en el Teensy
Reproducir audio a través del DAC del Teensy
Compilar la toolchain de GNU para Teensy
Display de 7 segmentos con interface serie en VHDL
Control de velocidad de un motor DC mediante lógica borrosa

Últimos Comentarios

Administrator (Avelino Herrera Morales)
22/04/15
Gracias, crack

Administrator (Avelino Herrera Morales)
26/11/14
Gracias Carlos. Atendí a la sugerencia que me hiciste y...

Administrator (Avelino Herrera Morales)
21/03/14

```
  pid.kp = kp;
  pid.ki = ki;
  pid.kd = kd;
  pid.sum = 0;
  pid.prevError = 0;
}

float pid_controller_run(struct pid_controller &pid, float error) {
  float p = pid.kp * error;
  pid.sum += error;
  float i = pid.ki * pid.delta * pid.sum;
  float d = pid.kd * (error - pid.prevError) / pid.delta;
  pid.prevError = error;
  return p + i + d;
}
```

Las pruebas empíricas realizadas han dado muy buenos resultados para:

$$K_p = K_i = K_d = 0.5$$

Con un período de muestreo $T = 0.01$. La inicialización, por tanto, quedaría así:

```
void setup() {
   ...
   pid_controller_init(motor_pid_controller, 0.01, 0.5, 0.5, 0.5);
   ...
}
```

Mientras que cada 10 milisegundos ($T = 0.01$) habrá que calcular el PID:

```
const float SET_POINT = 1600;   // consigna en rpm
unsigned long last_t = 0;

void loop() {
   unsigned long t = millis();
   if ((t - last_t) >= 10) {
      float error = SET_POINT - current_rpm;
      float u = pid_controller_run(motor_pid_controller, error);
      analogWrite(PWM_OUTPUT, (int) u);
      last_t = t;
   }
}
```

Pruebas realizadas

Para una consigna de 1000 rpm, la velocidad angular medida utilizando el PID es la siguiente (100ms entre medida y medida):

```
1013.99 rpm
1025.57 rpm
1013.03 rpm
1019.02 rpm
986.13 rpm
1003.95 rpm
1002.00 rpm
1013.65 rpm
999.07 rpm
977.64 rpm
1013.99 rpm
1037.49 rpm
1018.26 rpm
998.14 rpm
986.71 rpm
1006.64 rpm
1017.29 rpm
1017.43 rpm
```

Mientras que para una consigna de 1600 rpm, la velocidad angular medida fue la siguiente (100ms entre medida y medida):

```
1632.03 rpm
1591.01 rpm
1602.56 rpm
1583.28 rpm
1608.92 rpm
1578.28 rpm
1599.66 rpm
1583.61 rpm
1586.29 rpm
1616.21 rpm
1619.35 rpm
```

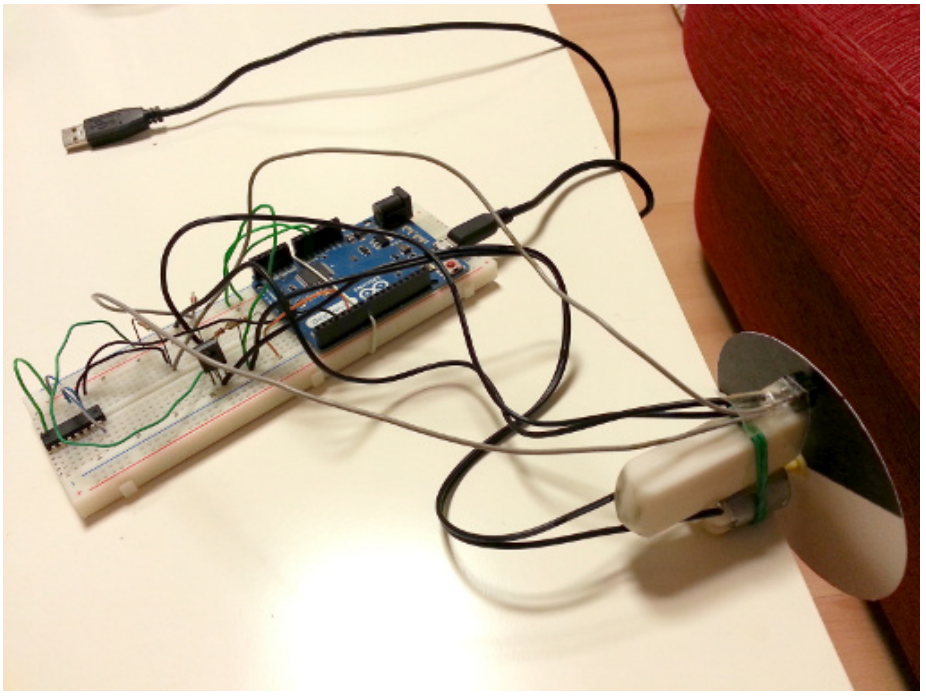
Hey, gracias por comentar, Carlos. El algoritmo se utiliza...

Administrator (Avelino Herrera Morales)
03/10/13
Efectivamente, como indicas al final, entiendo que el...

Administrator (Avelino Herrera Morales)
22/07/13
Si, claro, se podría (y se debería) encapsular. Con...

1594.39 rpm
1601.54 rpm
1581.11 rpm
1601.37 rpm
1606.68 rpm
1570.52 rpm
1602.39 rpm

El código fuente para Arduino puede descargarse de la sección **soft**.



[añadir comentario] (1034 visualizaciones) | **[0 trackbacks]** | **enlace permanente**

<< <Anterior | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **Siguiente> >>**