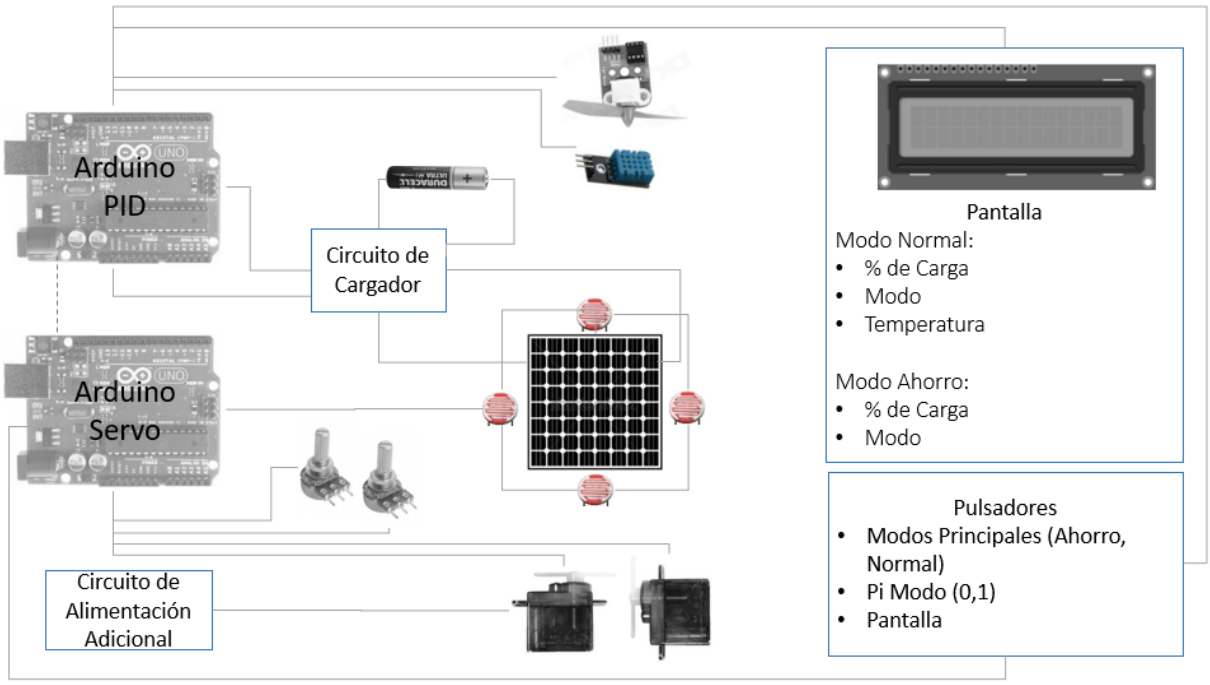
PROYECTO SEGUIDOR

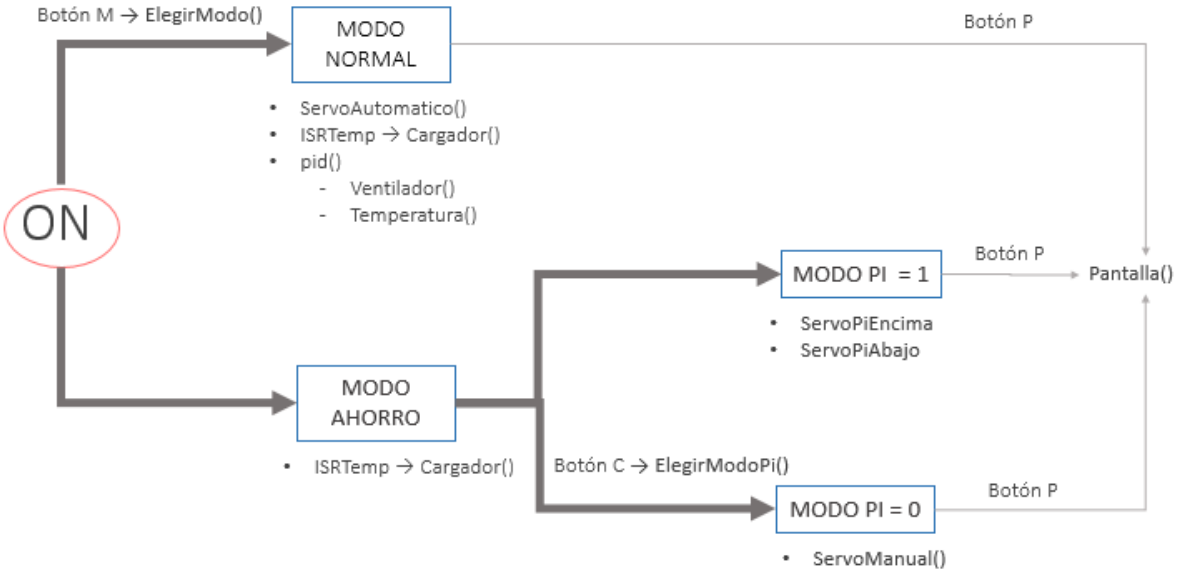
Yevheniya Kupchyk
MEM
2017

escuela técnica superior de ingeniería y diseño industrial

ÍNDICE

1. ESTRUCTURA GENERAL





1.1 MODO NORMAL

En Modo Normal, se asume que el ahorro de energía no es una prioridad. En este modo, las funciones de *ServoAutomático()*, y *pid()* funcionan mientras que la función *Cargador()* es llamada por una interrupción del temporizador T2. En este modo, el seguidor sigue a la luz, el ventilador reduce la temperatura del seguidor cuando sea necesario, y el “Arduino PID” compara los voltajes de la batería automáticamente para asegurar que el panel no continúa cargando a la batería, hasta ya esté cargada.

1.1.1. SERVO AUTOMÁTICO

1.1.1.1. UTILIZACIÓN

La parte principal del proyecto es un seguidor solar ejecutado por *ServoAutomatico()*. Se usa los fotoresistores en cada esquina del cartón que está puesto encima de dos servos. Usando la diferencia entre los datos que vienen de los fotoresistores, los servos se mueven, ajustando hasta que los fotoresistores de cada lado tenga mediciones parecidas.

1.1.1.2. COMPONENTES IMPORTANTES

Actuadores: **Servos**

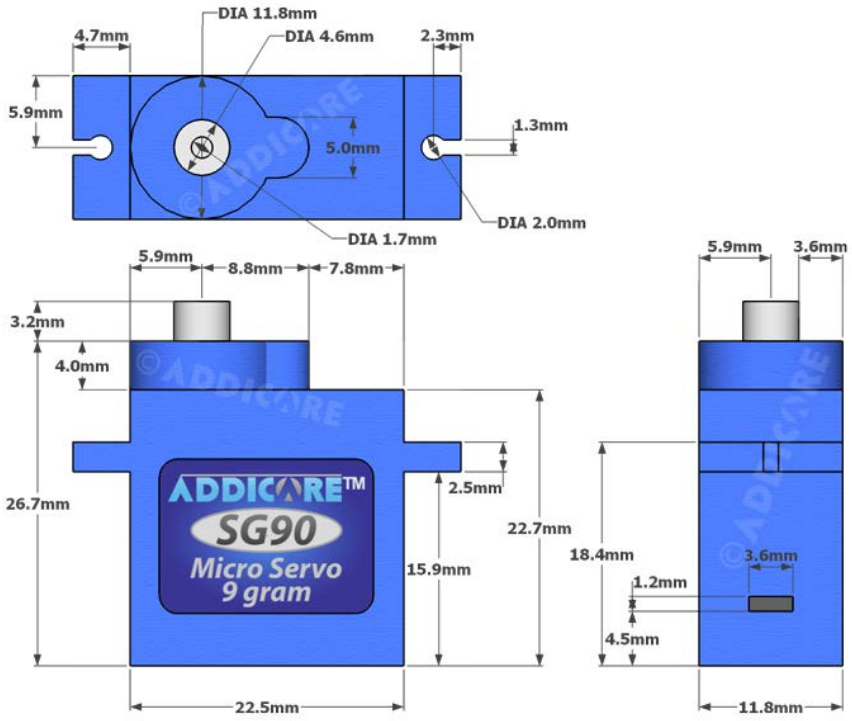


Figura 1.1: Estructura de hobby servo SG90
https://www.addicore.com/Addicore-SG90-Mini-Servo-p/113.htm

Un servo de este proyecto está conectado a una fuente de la alimentación externa por un cable naranja, a la tierra con el cable marrón, y a su pin digital correspondiente con el cable amarillo.

El modelo SG90 tiene engranajes plásticos, un motor DC, y un potenciómetro que facilita la retroalimentación según su circuito de control integrado.

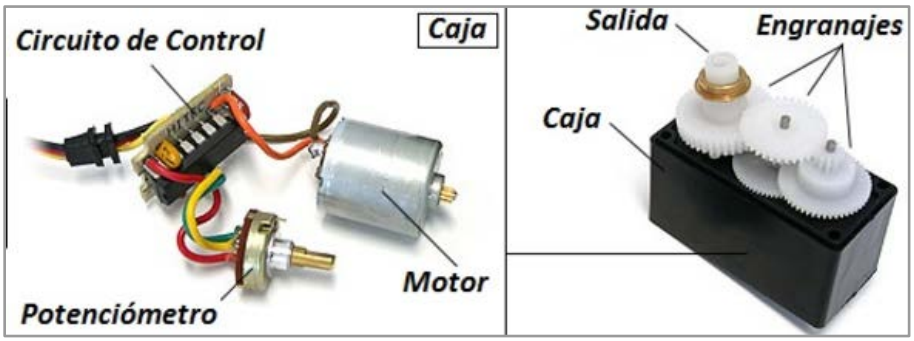


Figura 1.2: Componentes de un servo hobby

http://miArduinounotieneunblog.blogspot.com.es/2016/01/programar-posiciones-en-un-micro-servo.html

Similarmente al ventilador de este proyecto, mediante las señales de PWM se puede controlar el movimiento del servo. Una serie de pulsos cada 20 ms (50Hz) de anchuras variables (desde 1 a 2 ms, lo cual se traduce en un duty cycle de 5-10%) decide la dirección y la cantidad de movimiento. Por ejemplo, con un pulso de 1.5 ms un servo va a estar en la posición neutra de 90º, mientras que con un pulso de 1 ms se va a mover hacia 0º y con un pulso de 2 ms, el servo se moverá hacia 180º.

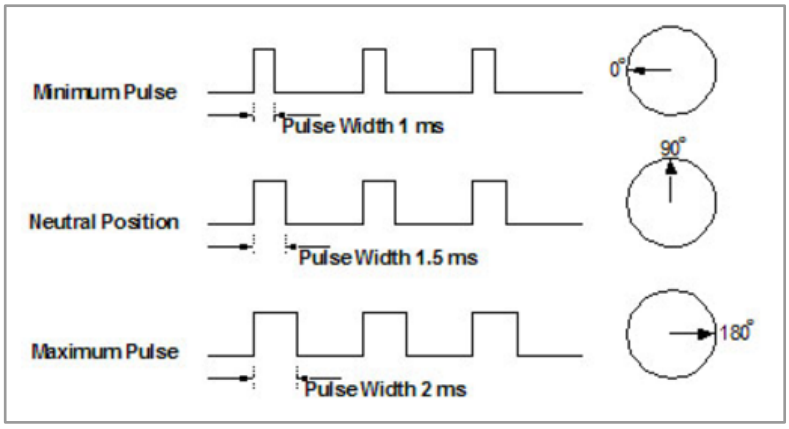


Figura 1.3: Movimiento de un servo segun la anchura de un pulso

https://www.servocity.com/how-does-a-servo-work

Es importante subrayar que los pulsos también pueden seguir una frecuencia variable (por ejemplo, cada 6 ms), es decir, pueden tener una frecuencia de refresco más alta, por ejemplo, entre 40 y 200 Hz.

Generalmente, el periodo de la señal PWM es de 20 ms y el pulso dura 1-2 ms. En este caso, para generar una señal correctamente, es necesario usar un temporizador de alta resolución, como el Temporizador 1 de Arduino Uno, que es de 16 bits. Si se usa una resolución de sólo 8 bits (que corresponde a $2^8 = 256$ posiciones), como por ejemplo en el caso del Temporizador

2, se notará que, a 5 V, para producir un duty cycle se necesita una resolución de 256*(20 ms), es decir, de más de 5000. Con un temporizador de 16 bits, el sistema tiene una resolución de $2^{16} = 65536$.

Un servo utiliza un control de bucle cerrado con un potenciómetro adjunto al eje de rotación. La anchura del pulso se mide por un circuito de control dentro de servo, que usa un amplificador de error para comparla con el ángulo de Set Point (SP). El SP (la posición deseada) se codifica en la anchura de pulso mientras que la señal del potenciómetro (PV = posición corriente) mide la posición del eje. En la etapa de potencia, se proporciona una corriente para mover los engranajes. El proceso se repite hasta que PV = SP o error = 0.

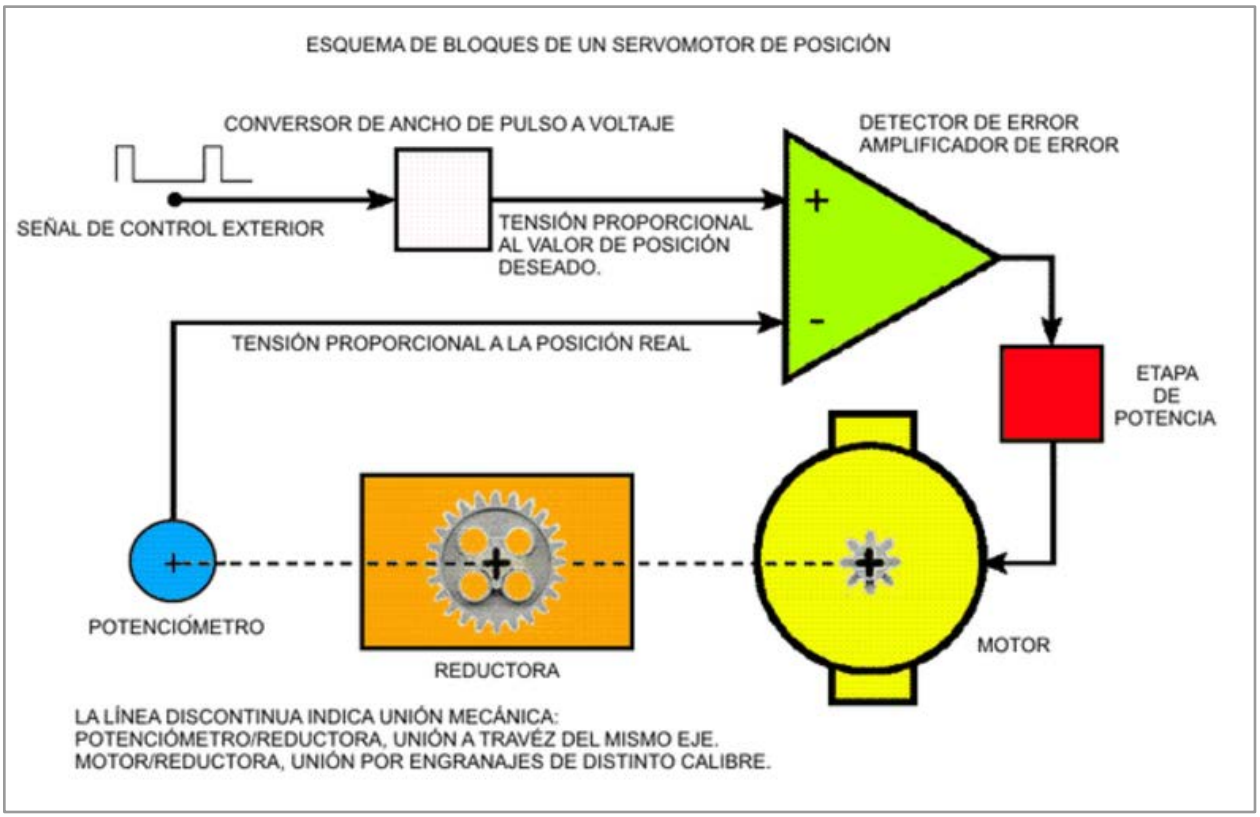


Figura 1.4: Ciclo del control de servo

http://beetlecraft.blogspot.com.es/2015/11/tutorial-servomotores-analogicos-y.html

Entradas Analógicas: **Fotoresistores**

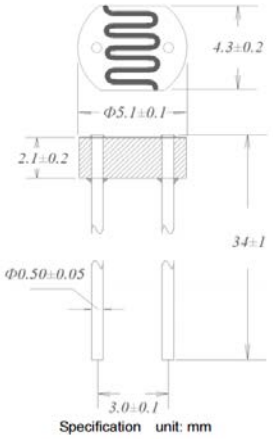


Figura 1.5: Medidas de un fotoresistor CdS (en mm)

http://akizukidenshi.com/download/ds/senba/GL55%20Series%20Photoresistor.pdf

Se usa un fotoresistor común de sulfuro de cadmio (CdS) de alta resistencia que responde al rango de la luz visible similarmente al ojo humano. Está compuesto de un cuerpo cerámico y dos electrodos (círculos pequeños en la imagen anterior) separados por CdS en forma de zigzag. Los fotones de la luz liberan los electrones almacenados, resultando en el movimiento de los electrones y los huecos creados en lados opuestos, y cuya densidad aumenta al aumentar la cantidad de luz. Se puede describir una relación entre las cantidades relativas de resistencias bajo 10 y 100 Lux:

$$\gamma = \frac{Lg(R10/R100)}{Lg(100/10)} = Lg(R10/R100) \longrightarrow$$

Specification	Light resistance (10Lux) (KΩ)	Dark resistance (MΩ)	γ_{10}^{100}	Response time (ms)		Illuminance resistance Fig. No.
				Increase	Decrease	
	5-10	0.5	0.5	30	30	2

Figura 1.6: Datos del fotoresistor usado

http://akizukidenshi.com/download/ds/senba/GL55%20Series%20Photoresistor.pdf

Se observa que el modelo usado tiene la resistencia de 5 a 10 KΩ al estar expuesto a 10 Lux después de ser expuesto a 400 - 800 Lux durante dos horas, así como una relación logarítmica entre las resistencias R10/R100 de 0.5.

Aparte de la disminución de la resistencia cuando aumenta la luz se observa también que la temperatura cambia la resistencia de los fotoresistores, lo cual significa que fotoresistores pueden diferenciar entre los niveles relativos de luz pero desafortunadamente no entre niveles absolutos sin calibración individual.

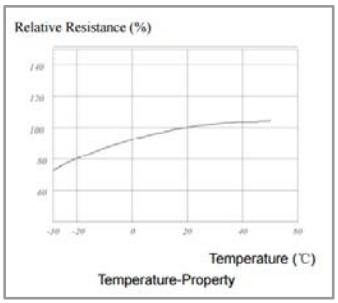
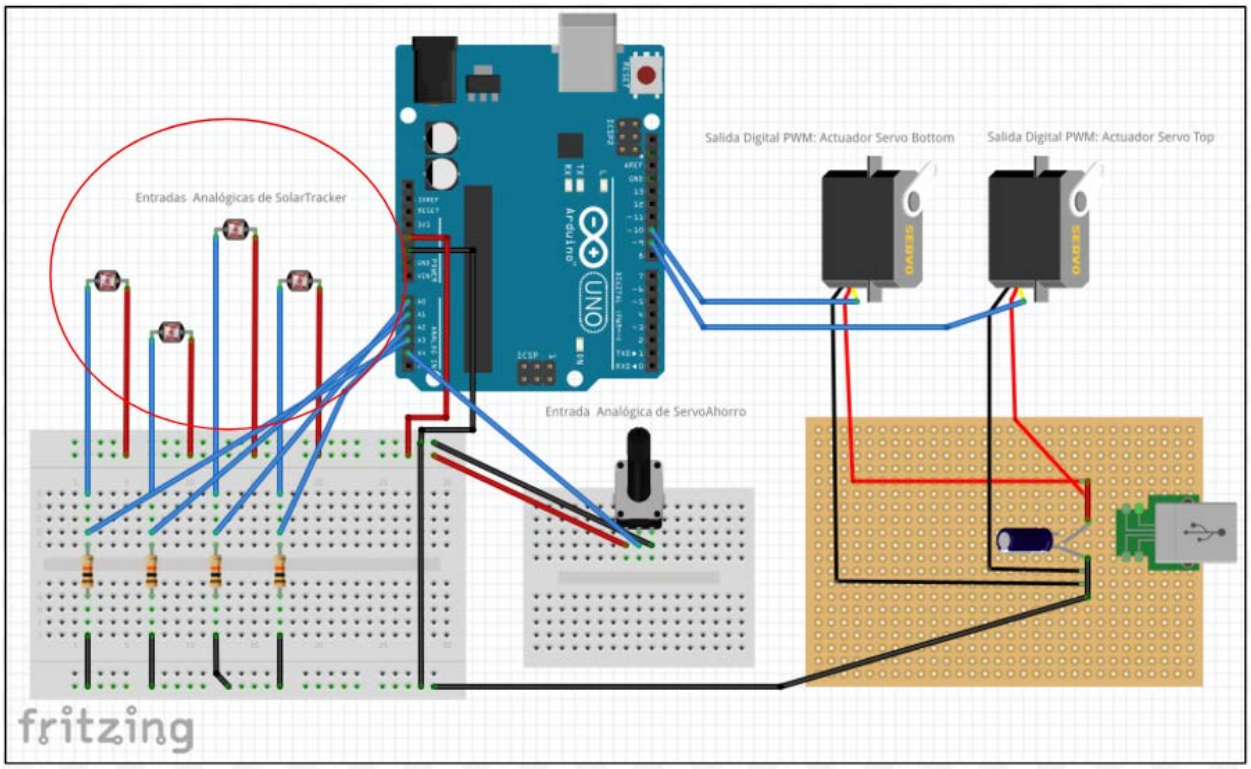


Figura 1.7: La relación entre la resistencia y la temperatura

http://akizukidenshi.com/download/ds/senba/GL55%20Series%20Photoresistor.pdf

1.1.1.3. MONTAJE



1.1.1.4. CÓDIGO

Después de conectar los pins de los fotoresistores, se mide la diferencia entre los lados opuestos del seguidor. Se establece una tolerancia del seguidor, lo cual determina la diferencia mínima entre los dos lados que causará un movimiento del seguidor.

```
void ServoAutomatico()
{
// OBTENER LECTURAS Y ESTABLECER LAS DIFERENCIAS ENTRE ELLOS QUE DECIDIRÁ LA DIRECCIÓN DEL MOVIMIENTO-------------

  // Leer los pins para obtener los datos de cada fotoresistor
  int Intensidad0 = analogRead(photo0);
  int Intensidad1 = analogRead(photo1);
  int Intensidad2 = analogRead(photo2);
  int Intensidad3 = analogRead(photo3);

  int tol = 20; //Determinar la diferencia que va a ser un punto después de que muevan los servos

  // Comparar los valroes en los lados opuestos (encima con abajo, derecho con izquierdo)
  int Dif01 = Intensidad0 - Intensidad1;
  int Dif23 = Intensidad2 - Intensidad3;
```

El código para el servo t (top/encima) y servo b (bottom/abajo) tienen la misma estructura pero distintos límites máximos de movimiento, par de fotoresistores, y comandos para escribir sus ángulos (bottom.write para servob y top.write para servot).

```
 // MOVER MOTOR ARRIBA (vertical)  MIENTRAS QUE HAY DIFERENCIA ENTRE ARRIBA Y ABAJO------------------------------
 if (-1*tol >  Dif23 || Dif23 > tol) //Asegurar que la diferrencia recibida entre arriba y abajo es significativa
 {
   if (Intensidad2 > Intensidad3) // Si el lado arriba recibe más luz que el lado abajo
   {
     servot = ++servot; // Mover el motor un unido hasta en la dirección de arriba solo hasta llegar a su límite
     if (servot > servotLimitHigh)
     {
       servot = servotLimitHigh; //No hacer nada, porque ya está en el limite
     }
   }
   else if (Intensidad2 < Intensidad3) // Si el lado arriba recibe menos luz que el lado abajo
   {
     servot= --servot; // Mover el motor un unido hasta en la dirección de abajo solo hasta llegar a su límite
     if (servot < servotLimitLow)
     {
       servot = servotLimitLow; //No hacer nada, porque ya está en el limite
     }
   }
   top.write(servot);
 }
```

```
  // MOVER MOTOR ABAJO (horizontal)  MIENTRAS QUE HAY DIFERENCIA ENTRE DER. y IZQ.---------------------------------
  if (-1*tol > Dif01 || Dif01 > tol) // Asegurar que la diferrencia recibida entre derecho y recto es significativa
  {
    if (Intensidad0 > Intensidad1)// Si lado izquierdo recibe más luz que el lado derecho
    {
      servob = ++servob; // Mover el motor un unido a la izquierda si todavía no llegó el servo a su limite
      if (servob < servobLimitLow)
      {
        servob = servobLimitLow; // No hacer nada, porque ya está en la limite
      }
    }
    else if (Intensidad0 < Intensidad1)
    {
      servob = --servob; // Mover el motor un unido a la derecha si todavía no llegó el servo a su limite
      if (servob > servobLimitHigh)//Si lado izquierdo recibe menos luz que el lado derecho
      {
        servob = servobLimitHigh; // No hacer nada, porque ya está en la limite
      }
    }
    bottom.write(servob);
  }
```

Después de identificar que la diferencia entre los lados es mayor que la tolerancia asignada se mueve el servo hacia el lado que recibe más luz (a no ser que el servo haya alcanzado su límite). Los límites se determinaron previamente por un proceso de calibración del servo y asignados antes del *setup().*

```
//VARIABLES------------------------------- LIMITES DE SERVOS
Servo top; // Servo arriba
int servot = 5;  // Posición de servo encima por defecto
const int servotLimitHigh = 80;
const int servotLimitLow = 5;

Servo bottom;   // Dervo abajo
int servob = 90;  // Posición de servo encima por defecto
const int servobLimitHigh = 180;
const int servobLimitLow = 0;
```

Durante el setup se usa *attach()* para conectar el nombre del servo a su pin correspondiente.

```
//SETUP SERVOS-----------------------------------------
top.attach(10);
bottom.attach(9);
```

Al ser llamada desde el loop (cuando MarcaModo = 1) por *ModoNormal()*, la función *ServoAutomatico()* se ejecuta continuamente, moviendo los servos un paso a la vez, produciendo un movimiento suave sin necesidad de usar PID.

```
//ELEGIR MODO-------------------------------------------------------------------------
if (MarcaModo == 1)
{
  ModoNormal();
}
else
{
  ModoAhorro();
}
```

```
void ModoNormal()
{
  // Llamar a ServoAutomático()-----------------------------------------------------
  ServoAutomatico();
}
```

1.1.2. PID

1.1.2.1. UTILIZACIÓN

El calor puede reducir el rendimiento de los paneles solares en un 15-25%. Por eso, se monta una estructura que simboliza el calentamiento por el sol y la respuesta del panel representado por el sensor de la temperatura enfriado por un ventilador cercano. Se usa un Gc dentro del bucle que contiene los parámetros de PI, aunque con mi sistema sólo un controlador con el control proporcional sería suficiente.

En realidad, se usaría un sistema de enfriamiento por agua, como en la figura abajo:

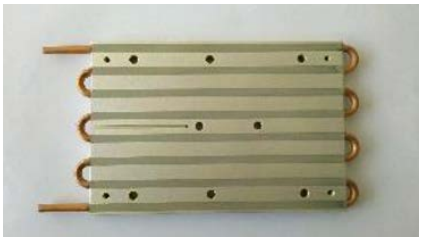


Figura 1.8: Placa de enfriamiento de un panel solar.

http://suzhouwint.sell.everychina.com/p-95002261-cold-plate-water-cooling-plate.html

1.1.2.2. COMPONENTES IMPORTANTES

Actuador: **Ventilador Keyes I9110**

Entrada Digital: **Sensor de Temperatura DHT11**

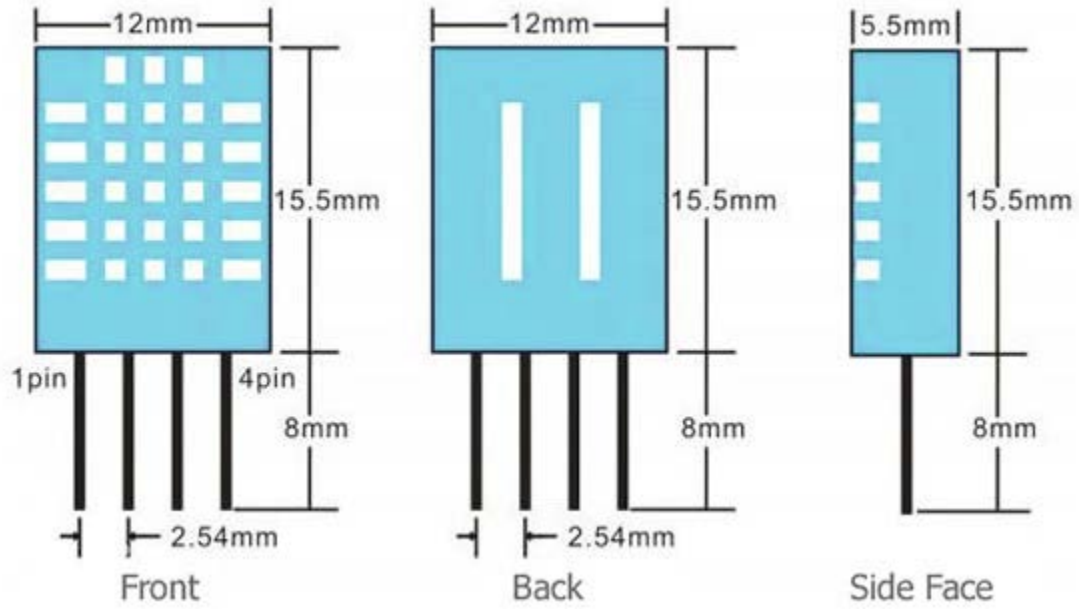


Figura 1.9: Medidas del sensor DHT11

https://tallerarduino.com/2012/12/24/sensor-dht11-humedad-y-temperatura-con-arduino/

DHT11 es un sensor termistor NTC cuya resistencia aumenta cuando baja la temperatura del aire porque aumenta la concentración de portadores.

Es importante mencionar que la variación de la resistencia con la temperatura no es lineal:

$$R_T = A \cdot e^{\frac{B}{T}} \text{ o bien } R_T = R_0(1 + \alpha T) \quad \text{donde } A = R_0 \cdot e^{-\frac{B}{T_0}}$$

R_T es la resistencia del termistor a la temperatura en K, R_o es la resistencia del termistor a la temperatura de referencia, α es el coeficiente de temperatura, y B es la temperatura característica del material. Se puede observar la falta de linealidad en una variación entre la temperatura y resistencia en la figura siguiente.

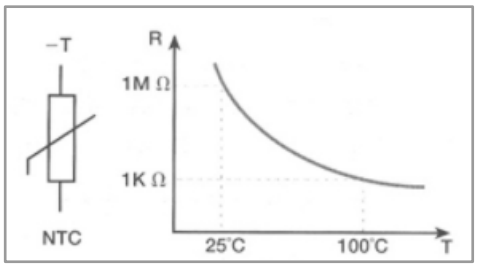
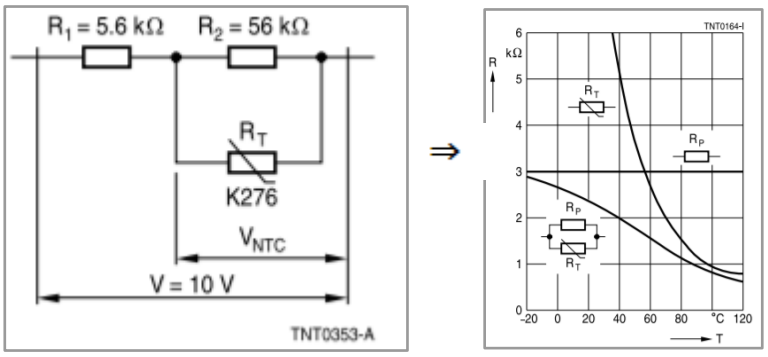


Figura 1.10: Relación entre resistencia y temperatura
http://agrega.juntadeandalucia.es/repositorio/25022013/69/

Por esta razón, al construir y definir los parámetros del funcionamiento es necesario elegir una zona que se comporta lo más linealmente posible y usar el sensor sólo en el rango indicado por el vendedor. En este proyecto, se usa entre 0ºC y 50ºC.

Es posible linealizar un sensor NTC al poner resistencias en paralelo o en serie con el sensor como en este ejemplo:



Se observa el cambio de la geometría de la curva con el uso de una resistencia en paralelo con el NTC.

Figuras 1.11: Gráficos de comportamiento del termistor
https://en.tdk.eu/download/531110/a3be527165c9dd17abca4970f507014f/pdf-applicationnotes.pdf

Se puede calcular la resistencia añadida tras una aproximación exponencial $R_P = R_T\left(\frac{B-2T}{B+2T}\right)$, que nos da la pendiente $\frac{dR}{dT} = -\frac{R_T}{\left(1+\frac{R_T}{R_p}\right)^2}\frac{B}{T^2}$.

Para empezar la comunicación, el microcontrolador manda una señal al sensor durante 18 ms con 20 - 40 µs de espera para que el sensor responda con el cambio del estado de ahorro al estado de funcionamiento, que tarda 80 µs. Después, el sensor manda los datos de la humedad y la temperatura al microcontrolador, mandando un pulso corto para codificar un 0 y un pulso largo para un 1.

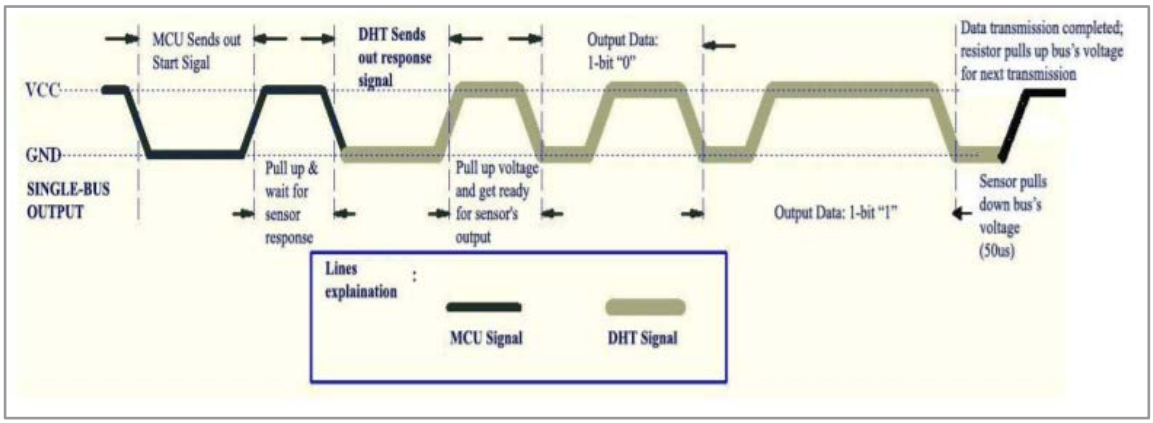


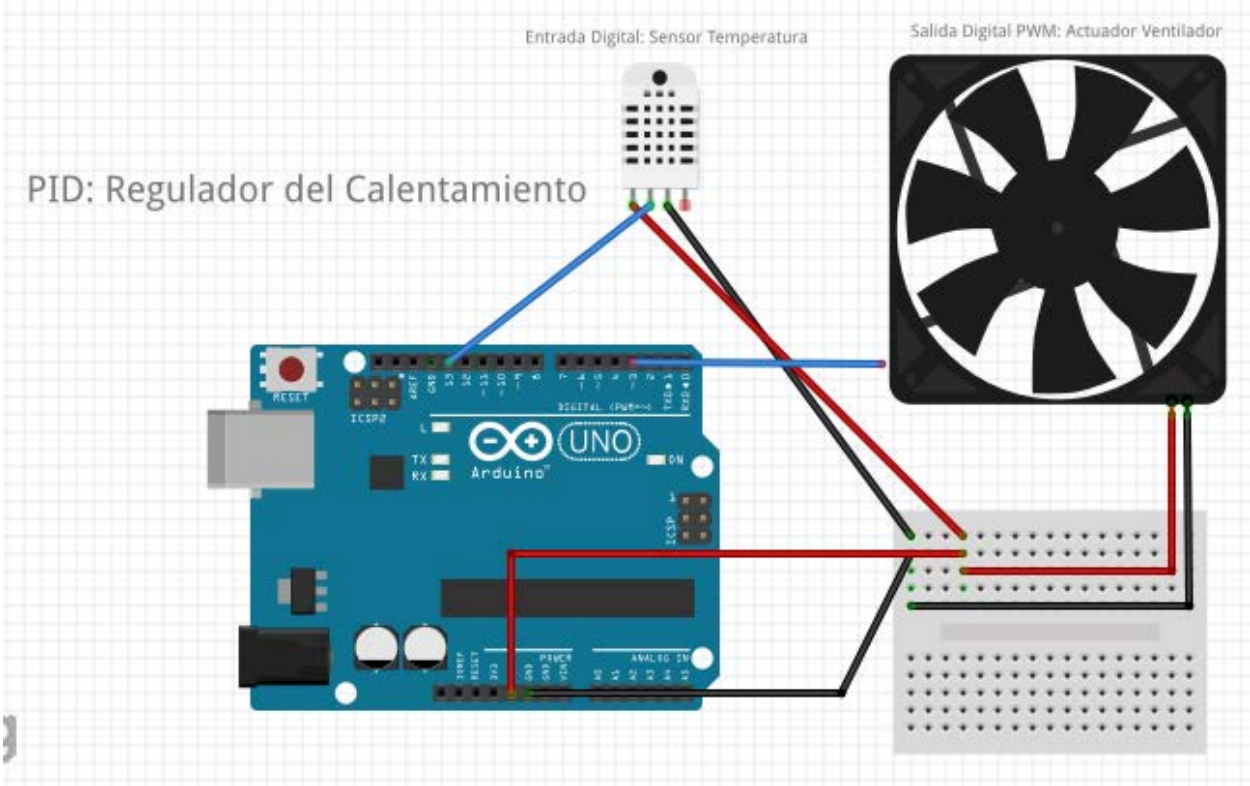
Figura 1.12: Ciclo de funcionamiento del sensor DHT11.

http://www.micropik.com/PDF/dht11.pdf

El inconveniente de este tipo de sensor es que, con más linealización, hay más pérdida de sensibilidad y que, pasado cierto punto, el autocalentamiento puede ocurrir, lo cual puede causar medidas erróneas. Además, con el sensor DHT11 se tiene que esperar un par de segundos para detectar un cambio de temperatura, lo cual puede ser perjudicial en un sistema rápido.

En el caso de este proyecto, el sensor se usa en un sistema de muy baja robustez y rapidez. El sensor sólo sirve bien para demostrar el control en bucle cerrado entre el ventilador de baja potencia y temperaturas de bajo rango (20-40ºC).

1.1.2.3. MONTAJE

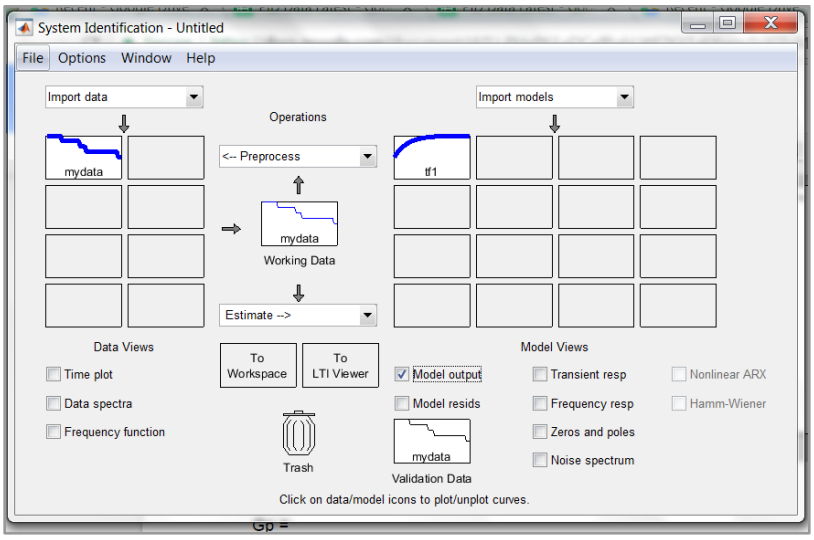


Se sigue estos pasos para encontrar Gp y Gc:

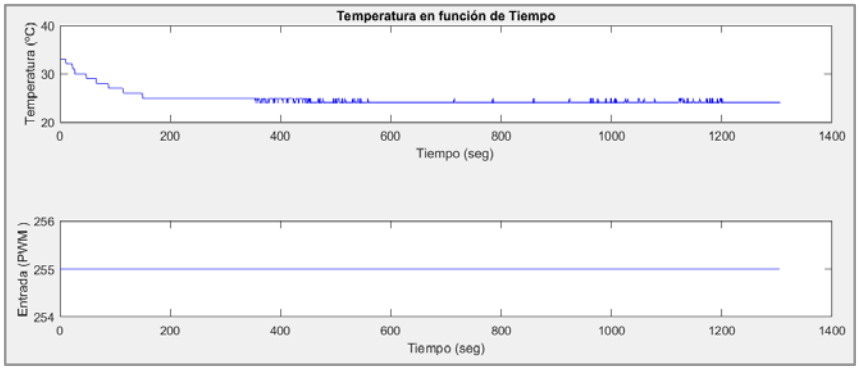
1. Calentar el sensor con una lámpara a una distancia fija hasta estabilizar temperatura a 35ºC.
2. Reiniciar programa con ventilador encendido a PWM máximo.
3. Dejar que se enfríe el sensor hasta estabilizar la temperatura.
4. Copiar los datos del monitor serie a Excel.
5. Importar los datos a Matlab. *colTime* contiene los tiempos, *colTemp* las temperaturas, y *colPWM* contiene el valor PWM usado (255).

```
>> pidData = xlsread('YevPIDBeforeExam.xlsx');
>> pidData = xlsread('YevPIDBeforeExam.xlsx');
>> colTime= pidData(:,1)
>> colTemp= pidData(:,2)
>> colPWM= pidData(:,3)
```

6. Se usa PID Identification App en Matlab. Después de la importación de los datos se obtiene una representación gráfica.



7. Se usa Time plot para mostrar la temperatura



8. Elegir una estimación de primer grado de la forma general:

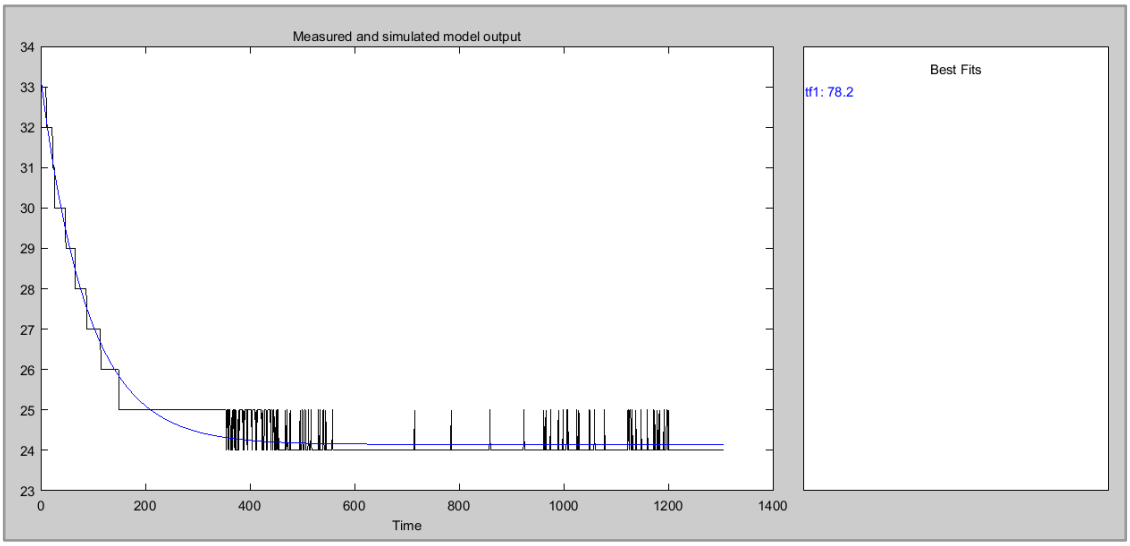
$$Gp = \frac{Ke\ (1 + Tn * s)}{(1 + Td * s)}$$

He probado con otras formas, pero otras estimaciones con más grados de libertad no mejoran la calidad del modelo significativamente.

9. Recibir la representación de mis datos:

```
tf1 =
  From input "u1" to output "y1":
  0.1298 s + 0.001062
  -------------------
     s + 0.01122

Name: tf1
Continuous-time identified transfer function.

Parameterization:
   Number of poles: 1   Number of zeros: 1
   Number of free coefficients: 3
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on time domain data "mydata".
Fit to estimation data: 78.2% (stability enforced)
FPE: 0.1202, MSE: 0.1195
```

10. Recibir la representación gráfica de mi resultado (azul) con mis datos (negro).



11. Obtener la información del comportamiento de mi sistema.

```
>> stepinfo(tf1)
ans =
  struct with fields:

       RiseTime: 195.8153
   SettlingTime: 348.6763
    SettlingMin: 0.0947
    SettlingMax: 0.0980
      Overshoot: 37.1316
     Undershoot: 0
           Peak: 0.1298
       PeakTime: 0
```

12. Obtener los parámetros que corresponden a mi objetivo. En mi caso, idealmente, quiero aumentar la rapidez de mi sistema.

    Objetivo: Reducir tr (Rise Time) y ts (Settling Time) si es posible.

Es importante subrayar que muchas veces lo que es posible teóricamente, no es siempre posible en realidad.

tr corriente = 195.8153 seg reducir a tr objetivo = 100 seg
ts corriente = 348.6763 seg reducir a ts objetivo = 200 seg

$$\left.\begin{array}{l} t_r = \dfrac{\pi - arccos(\xi)}{w_n\sqrt{1-\xi^2}} = 100 \Rightarrow w_n = \dfrac{\pi - arccos(\xi)}{100\sqrt{1-\xi^2}} \\ t_s = \dfrac{3}{\xi w_n} = 200 \Rightarrow \xi = \dfrac{3}{200 w_n} \end{array}\right\}$$

Resolver para obtener w_n y ξ ⇒
⇒ ξ = 0.5678, w_n= 0.0264
parámetros de un sistema más rápida

13. Calcular analíticamente un Gc usando ξ y w_n basada en mis objetivos.

$$\left.\begin{array}{l} Gc = K(1 + \dfrac{1}{Ti * s}) \\ Gp = \dfrac{Ke(1+Tn*s)}{(1+Td*s)} \end{array}\right\} \quad Gcc = \frac{K(1+\frac{1}{Ti*s})\frac{Ke(1+Tn*s)}{(1+Td*s)}}{1+K(1+\frac{1}{Ti*s})\frac{Ke(1+Tn*s)}{(1+Td*s)}} =$$

$$= \frac{K\left(1+\frac{1}{Ti*s}\right)\frac{Ke(1+Tn*s)}{(1+Td*s)}}{(1+Td*s)+K\left(1+\frac{1}{Ti*s}\right)Ke(1+Tn*s)} = \frac{K\left(1+\frac{1}{Ti*s}\right)Ke(1+Tn*s)}{(1+Td*s)+K\frac{(Ti*s+1)}{Ti*s}Ke(1+Tn*s)}$$

$$= \frac{K\frac{(Ti*s+1)}{Ti*s}Ke(1+Tn*s)}{\frac{Ti*s(1+Td*s)+K(Ti*s+1)Ke(1+Tn*s)}{Ti*s}}$$

$$= \frac{numerador}{Ti*s(1+Td*s)+K(Ti*s+1)Ke(1+Tn*s)}$$

$$\Rightarrow Ti*s+TiTd*s^2+(KTi*s)(Ke+KeTn*s)$$

$$= Ti*s+TiTd*s^2+KTiKe*s+KTiKeTn*s^2+KKe+KKeTn*s$$

$$= (TiTd+KTiKeTn)*s^2+(Ti+KTiKe+KKeTn)*s+K$$

$$= s^2+\frac{(Ti+KTiKe+KKeTn)}{(TiTd+KTiKeTn)}*s+\frac{KKe}{TiTd+KTiKeTn}$$

Con esta estructura se puede comparar mi resultado con mis objetivos:

$$s^2 + 2\xi w_n * s + {w_n}^2 = s^2 + 2(0.5678)(0.0264) * s + (0.0264)^2$$

$$\left.\begin{array}{l} \dfrac{(Ti+KTiKe+KKeTn)}{(TiTd+KTiKeTn)} = 2(0.5678)(0.0264) \\ \dfrac{KKe}{TiTd+KTiKeTn} = (0.0264)^2 \end{array}\right\}$$

Resolver ⇒ $K = -8.073, Ti = 256.7$

Se observa que el valor de K es negativo, lo cual significa que, con más entrada en mi actuador, mi respuesta (la temperatura) baja. Se observa también que el Ti es desproporcionadamente alto lo cual puede resultar en demasiadas oscilaciones y

potencialmente reducir la estabilidad de mi sistema. Con pruebas físicas de mi sistema, puedo probar este valor y ajustar mi Ti.

14. Poner la Gc encontrada en Matlab con la Gp original usando modo *tf.*

```
>> Gp = tf([0.1298 0.001062], [1 0.01122]);
Gp =
  0.1298 s + 0.001062
  -------------------
     s + 0.01122

>> K=-8.073;

>> Ti=256.7;

>> Gc = tf([K*Ti K],[Ti 0])
Gc =
  -2072 s - 8.073
  ---------------
     256.7 s
```

15. Juntar Gc y Gp en un bucle.

```
>> Gcc = feedback(Gp*Gc,1)

Gcc =

   269 s^2 + 3.249 s + 0.008574
  ---------------------------------
  12.29 s^2 + 0.3685 s + 0.008574

Continuous-time transfer function.
```

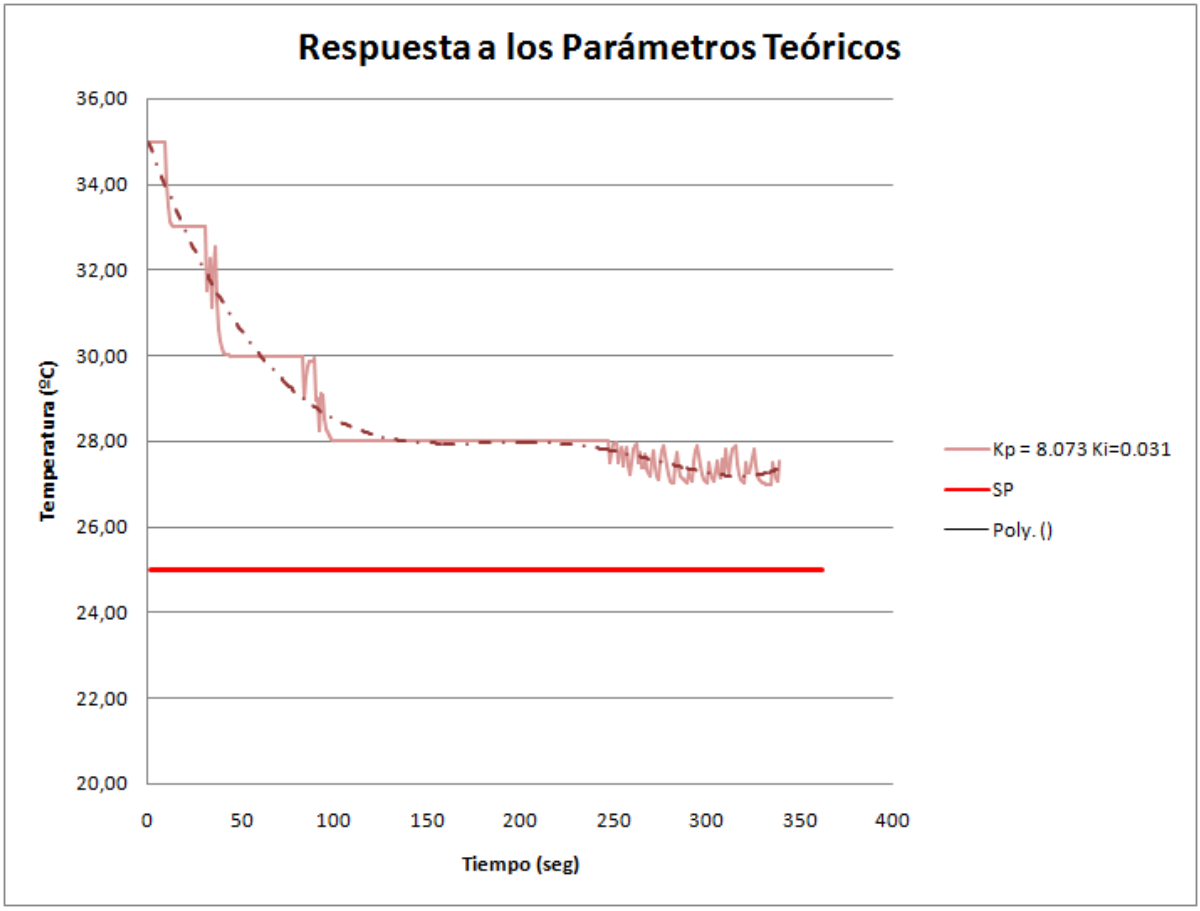
16. Obtener la información del comportamiento del bucle cerrado con control.

```
>> stepinfo(Gcc)
ans =
  struct with fields:

       RiseTime: 48.8392
   SettlingTime: 192.2942
    SettlingMin: -2.4124
    SettlingMax: 2.8095
      Overshoot: 2.0888e+03
     Undershoot: 241.2366
           Peak: 21.8876
       PeakTime: 0
```

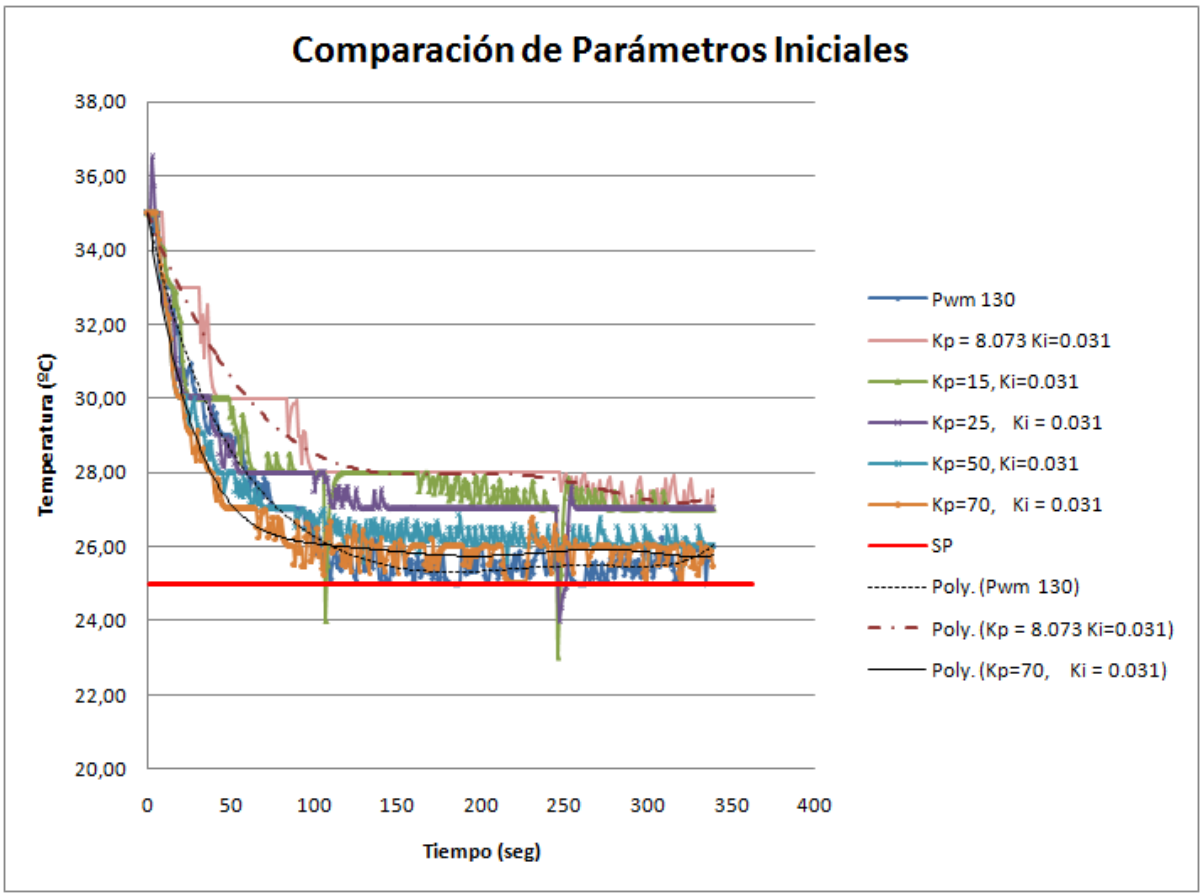
```
>> damp(Gcc)

      Pole              Damping     Frequency     Time Constant
                                  (rad/seconds)     (seconds)

-1.50e-02 + 2.17e-02i   5.68e-01     2.64e-02       6.67e+01
-1.50e-02 - 2.17e-02i   5.68e-01     2.64e-02       6.67e+01
```

Se usa los valores de Kp = 8.073 y Ki = 0.031 obtenidos por el método teórico para obtener la curva de respuesta real que está representada abajo, y aproximada por un polinomio de

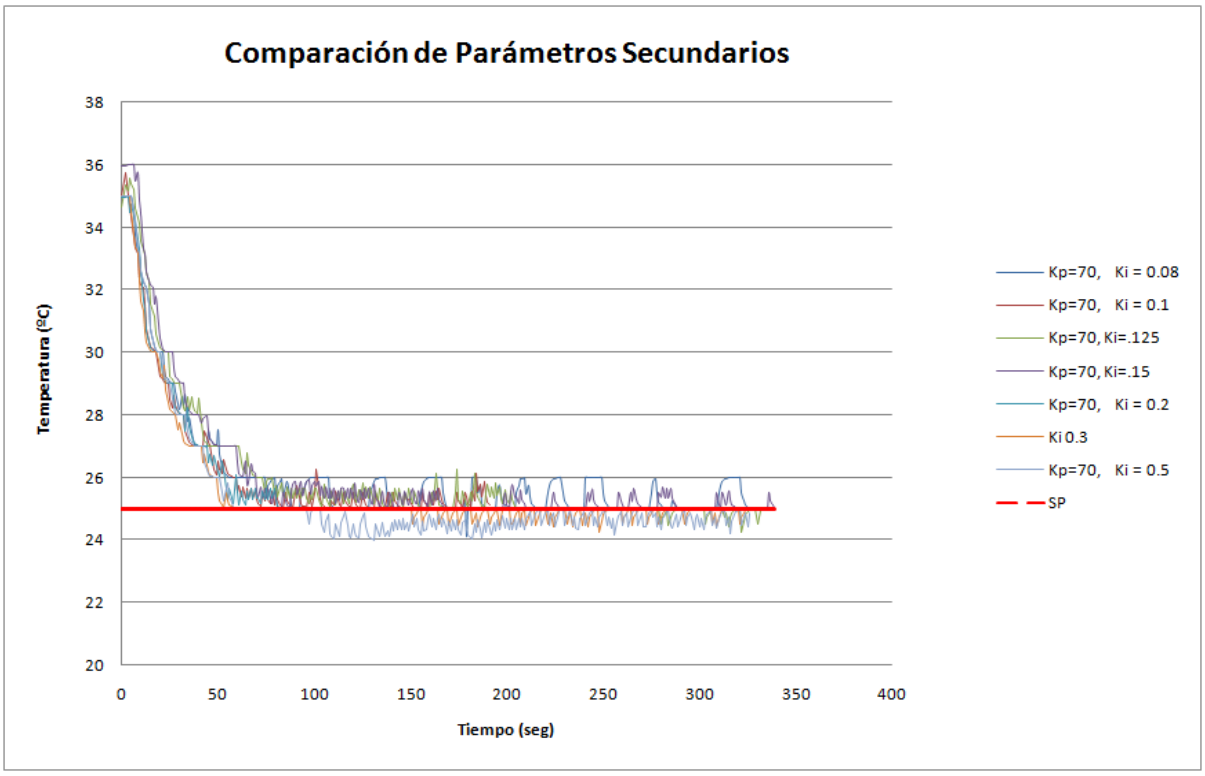
6 grados dentro de Excel (se usa sólo para una comparación general entre otros parámetros). Se observa que con los parámetros teóricos no se puede llegar a mi temperatura deseada. Esto no significa que el método teórico no funciona, sino que el ventilador simplemente no tiene la potencia para causar una diferencia de temperatura rápidamente. Así, se observa que el sistema de la figura es demasiado lento.



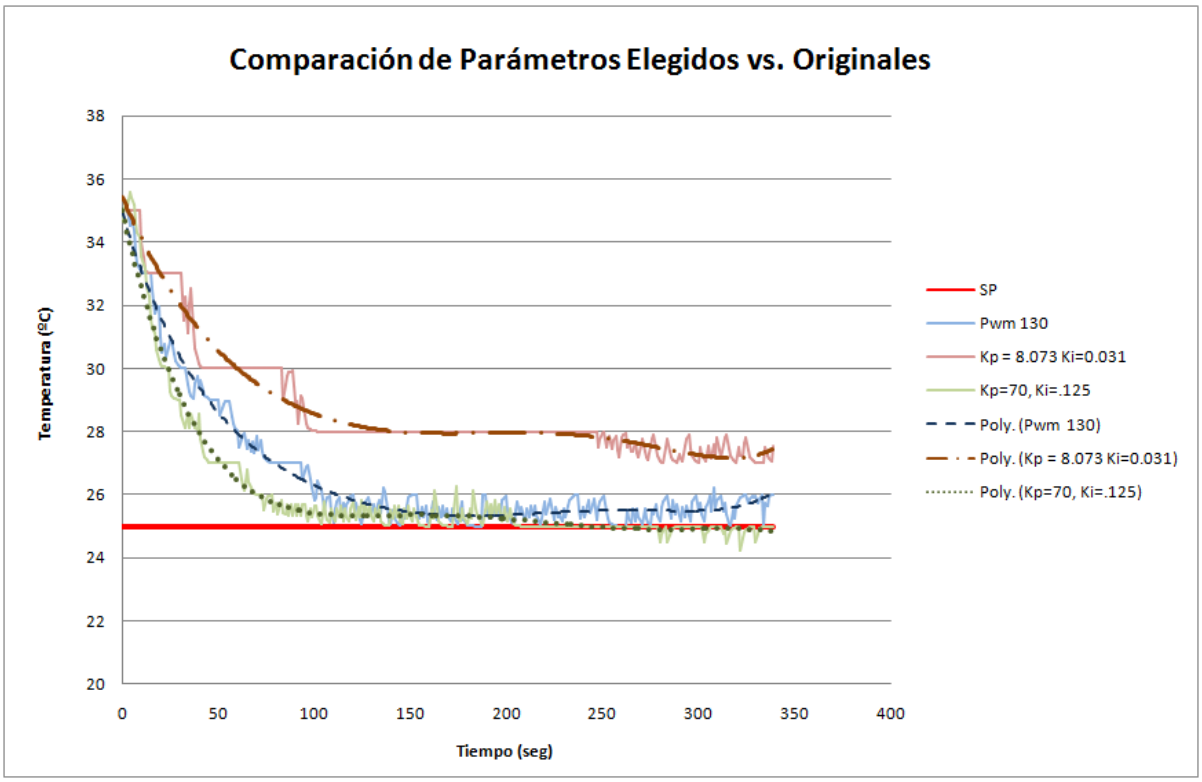
Basado en los parámetros teóricos se hace una multitud de pruebas. Se enciende el ventilador, y se usa una velocidad media para observar una respuesta preliminar. Sería desaconsejable siempre usar la potencia máxima para enfriar el proceso para no dañar al actuador. Usando la Ki teórica, se compara respuestas distintas de Kp. Se observa que los Kps más altos dan respuestas más rápidas.



Usando el valor máximo de Kp se compara los valores de Ki alrededor del valor dado por el método teórico. Se observa que tras cierto punto (como por ejemplo en el caso de Ki = 0.5), ocurren sobreoscilaciones en la respuesta. En realidad, el actuador nunca para de encenderse y apagarse, lo cual puede reducir su vida útil.



Se elige el valor que redujo las sobreoscilaciones y que consigue cambiar suavemente la temperatura del sensor. Usando el valor de Kp = 70 y Ki = 0.124, se puede comparar la respuesta a la original y sin control.



Se observa que los parámetros elegidos dan una respuesta más rápida y precisa que la teórica y sin control. Es interesante que ambos parámetros elegidos con práctica son diez veces más que los parámetros teóricos lo cual puede significar que las proporciones dadas por la teoría no son equivocadas, sino que el ventilador tiene tan baja potencia y tan mala respuesta que es necesario aumentar ambos parámetros a la vez para añadir la rapidez necesaria como se observa en la figura anterior.

1.1.2.4. CÓDIGO

Antes de *setup()* se convierte un pin analógico a un pin digital (15) y se conecta al sensor de temperatura. La conversión es necesaria para usar un pin digital adicional libre dentro del "Arduino PID" para mandar comandos al "Arduino Servo".

Se inicializa el PID, utilizando los valores de la parte anterior en el modo REVERSE, lo cual significa que al aumentar la potencia del actuador, mi variable medida tiene que bajar, así que no tengo que utilizar la forma negativa de K previamente obtenida.

```
// VARIABLES-----------------------------------PID
#define tempPin 15   // Antes era 13
float temperatura = 0;
double consKp = 70, consKi=0.125, consKd=0; //Definir Tuning Parameters
double Input, Output, Setpoint;
PID myPID(&Input, &Output, &Setpoint, consKp, consKi, consKd, REVERSE);
```

```
// VARIABLES----------------------------------VENTILADOR
#define pinVentilador 5
```

Dentro de *setup()* se asigna el modo OUTPUT al pin del ventilador, se da los límites al PID y se elige el modo MANUAL para no tener el PID encendido automáticamente al encender el Arduino PID.

```
// SETUP PID----------------------------------------------------------------
pinMode(pinVentilador,OUTPUT); // Ventilador
myPID.SetOutputLimits(0,255); // Crear los limites de PID
myPID.SetMode(MANUAL); // No encender PID
```

Dentro de *loop()*:

```
  // ELEGIR MODO--------------------------------------------------------------------------
  if (MarcaModo == 1) // Decide el modo de funcionamiento del Arduino PID y el Arduino Servos
  {
    ModoNormal();
    digitalWrite(pinControlModoArduinoServo, HIGH);  // Manda el mensaje al pin del Arduino Servos para cambiar el modo
  }
  else
  {
    ModoAhorro();
    digitalWrite(pinControlModoArduinoServo, LOW);
  }
 //Serial.println(MarcaModo);
}
```

Se elige el modo AUTOMATIC para automáticamente encender PID justo al entrar en *ModoNormal().*

```
void ModoNormal()
{
 // LLAMAR PID()-------------------------------------------------------------------------
  myPID.SetMode(AUTOMATIC);

  Ventilador(pid());
}
```

Dentro de *ModoNormal()* se llama a la función *Ventilador(),* que usa el valor proporcionado por la función *pid().*

```
// IMPORTANTE: El ventilador funciona al revez con un pequeño márgen de funcionamiento:
//             pwm de 190 es la velocidad mínima mientras que 0 es la máxima

void Ventilador(int Speed)
{
 // USAR PWM PARA CONTROLAR EL VENTILADOR-----------------------------------------------
 if (Speed > 190)
 {
  digitalWrite(pinVentilador, HIGH);
 }
 else
  analogWrite(pinVentilador, Speed); //(pin, speed) // 190 = velocidad mínimna
 }
```

La función *pid()* usa myPID para calcular el valor de PWM que el ventilador tiene que recibir según los parámetros de Kp y Ki previamente dados.

```
long int tref_temperatura = millis();

int pid()
{
  if (millis() - tref_temperatura >= 1000) // Para evitar el uso de delay, y dejarlo sólo cuando no hay otro opción
  {                                        // se usa millis() para reducir la frequencia de leída
    Input = Temperatura(); // Llamar Temperatura() que es el Input al PID
    tref_temperatura = millis();
  }

  Setpoint = 32; // Setpoint del PID
  myPID.Compute(); // Calcular output del PID
  int Speed = map(Output, 255, 0, 0, 190); // Dar el valor que el ventilador entiende. Usando Keyes Ventilador,
                                           // 0 PWM = Max velocidad y 190 = Parada

  return Speed; // Devolver Speed, lo que alimenta Ventilador()
}
```

La función *pid()* también usa la función *Temperatura()* para obtener el PV, basándose en un método simple para suavizar los datos leídos del sensor y para reducir conmutaciones del ventilador. Hay veces cuando el sensor empieza a leer valores muy grandes (100, etc.) que desestabilizan el PID añadiendo un error muy grande. Para evitar este problema, si el sensor lee una temperatura completamente fuera de su rango, se usa un delay pequeño para darle tiempo para estabilizarse.

```
float Temperatura()
{
// USAR EL SENSOR DHT11 PARA OBTENER TEMPERATURA---------------------------------------------
DHT.read(tempPin); // Inicializar leer temperatura
temperatura = DHT.temperature; // Leer la temperatura
if (temperatura > 50) // Se el ruido causa leidas enormes, se usa delay para esperar hasta que estabeliza el sensor
 {
   delay(50);
 }
 else
 {
   Input = .5*temperatura + 0.5*Input; // Un filtro para aumentar suavidad de leida de la temperatura.
 }                                     // Esto también reduce conmutaciones del actuador
 return Input;
}
```

1.2. MODO AHORRO

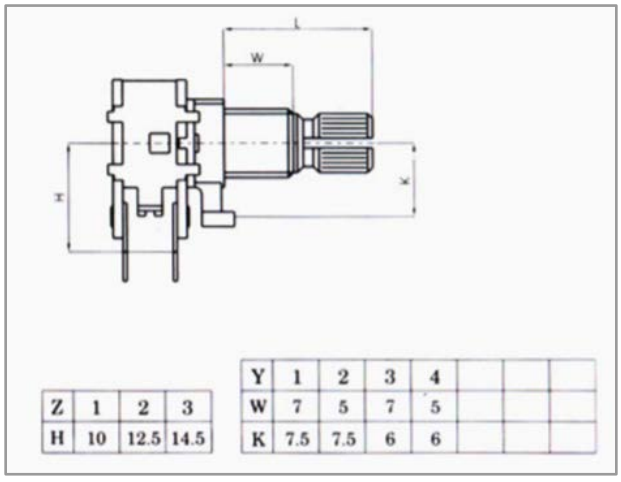
1.2.1. SERVO AHORRO - MarcaPi 0

1.2.1.1. UTILIZACIÓN

En el Modo Ahorro, los servos no utilizan los fotoresistores para seguir la luz. Por defecto, en el Modo Ahorra, la MarcaPi = 1, lo cual significa que el Arduino Servo siempre está esperando a los comandos del Arduino PID. Pero con el botón C se lanza una interrupción de hardware que cambia MarcaPi a 0, que permite utilizar los potenciómetros para controlar los servos.

1.2.1.2. COMPONENTES IMPORTANTES

Entrada Analógica: **Potenciómetro**



Z	1	2	3
H	10	12.5	14.5

Y	1	2	3	4			
W	7	5	7	5			
K	7.5	7.5	6	6			

Figura 1.13: Esquema de un potenciómetro (en mm).

https://www.sparkfun.com/datasheets/Components/General/R12-0-.pdf

Una resistencia variable rotativa está compuesta por un dispositivo móvil que se puede mover a lo largo de una resistencia, variando la longitud del tramo de pista con el que se está en contacto lo cual resulta en una resistencia que cambia.

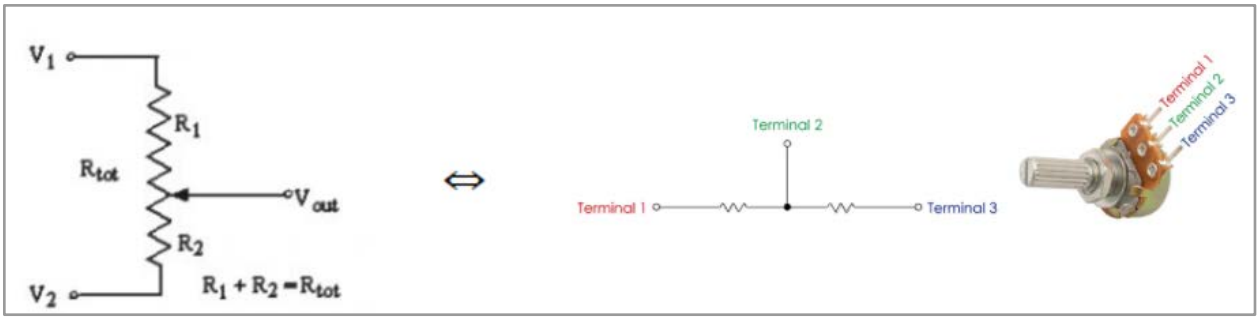
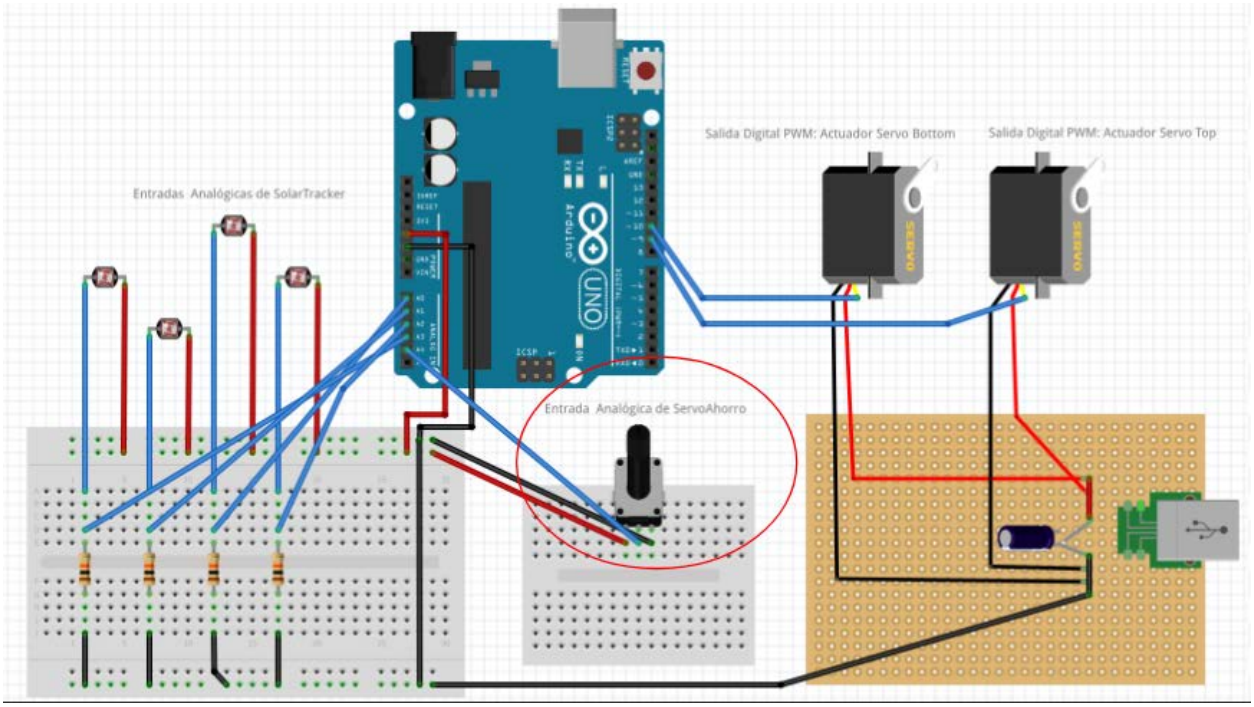


Figura 1.14: Circuito que corresponde al potenciómetro usado.

http://panamahitek.com/que-es-y-como-funciona-un-potenciometro/

Las entradas analógicas de Arduino pueden leer valores entre 0 y 1023 para un voltaje que varía entre 0 y 5 V, es decir, tiene una resolución de 0.004 V. De esta forma, la resistencia mínima corresponde al valor máximo del voltaje mientras que la resistencia máxima corresponde a su valor mínimo.

1.2.1.3. MONTAJE



1.2.1.4. CÓDIGO

Antes del *setup()* se conectan los potenciómetros a los pins analógicos correspondientes.

```
//VARIABLES------------------------------- SERVO MANUAL
int valorHorizontal;
#define potenciometroTop A5
#define potenciometroBottom A4
int potenciometroValorTop;
int potenciometroValorBottom;
```

Durante el *loop()*, al igual que en el caso de *ServoAutomatico()*, se llama a la función correspondiente según el modo elegido.

```
//ELEGIR MODO---------------------
if (MarcaModo == 1)
{
  ModoNormal();
}
else
{
  ModoAhorro();
}
```

Si MarcaPi = 0, *ModoAhorro()* llama a *ServoManual()*.

```
void ModoAhorro()
{
  // Llamar a ServoAhorro()-----------------------------------------------
  if (MarcaPi == 0)
  {
    ServoManual();
  }
  else
  {
    // Reciben pulsos de Servo PID y cambian los ángulos correspondientes
    ServoPiAbajo();
    ServoPiEncima();
  }
}
```

Se usa *map()* para interpolar la lectura del potenciómetro de 1023-0 a un valor entre los ángulos límites de cada servo que corresponde a un ángulo de un servo. Usando dos potenciómetros se manda y lee los ángulos de los dos servos en Modo Ahorro cuando MarcaPi es igual a 0.

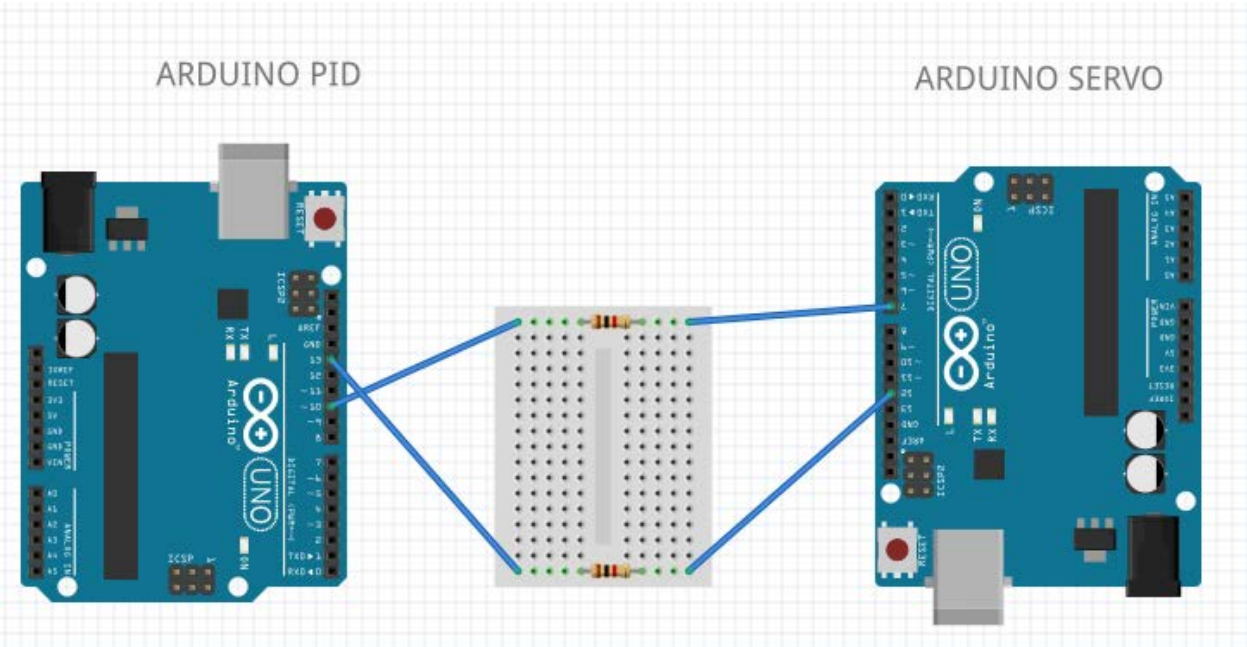
```
void ServoManual()
{
  // USAR ENTREADA ANALÓGICA PARA MOVER EL SERVO----------------------------------------------------
  potenciometroValorTop = analogRead(potenciometroTop); // Leer los datos del potenciometro
  servot = map(potenciometroValorTop, 0, 1023, 5, 80); // Asgignar los datos leido entre los
                                                       //limites del servo que está encima
  top.write(servot);  // Mover el servo hasta llegar al valor indicado en el paso anterior

  potenciometroValorBottom = analogRead(potenciometroBottom); // Leer los datos del potenciometro
  servob = map(potenciometroValorBottom, 0, 1023, 0, 180); // Asgignar los datos leido entre los
                                                           //limites servo que está abajo.
  bottom.write(servob); // Mover el servo hasta llegar al valor indicado en el paso anterior

  Serial.println(potenciometroValorTop);
  Serial.println(potenciometroValorBottom);
}
```

1.2.2. SERVO AHORRO - MarcaPi 0

1.2.2.1. UTILIZACIÓN

Se utiliza la MarcaPi para mandar comandos desde la Raspberry Pi (RPi) a “Arduino PID” que consecuentemente manda los comandos al “Arduino Servo” mediante pulsos. Por no tener más pins de PWM disponibles en ambos Arduinos, he creado mi propio pulso en Arduino PID usando un pin digital normal, cuya anchura codifica un ángulo. El Arduino Servo, que está siempre esperando los comandos de RPi, lee la anchura y la decodifica en un ángulo.

1.2.2.2. MONTAJE



1.2.2.3. CÓDIGO

Antes del setup()

```
// VARIABLES-------------------------------PI MODO REMOTO
#define pinControlModoArduinoServo 9
#define pinControlServoAb 10
#define pinControlServoEn 13
int contadorPi = 0;
int *contadorPiPointer;
int MarcaCambiarAngEncima = 0;
int MarcaCambiarAngAbajo = 0;
```

Durante el setup()

```
// SETUP PI MODO REMOTO-----------------------------------------------
pinMode(pinControlServoAb,OUTPUT); // Pin que manda pulsos al Arduino Servo
pinMode(pinControlServoEn,OUTPUT); // Pin que manda pulsos al Arduino Servo
```

```
contadorPiPointer = &contadorPi;
countDisconnectPointer = &countDisconnect;
```

Se comunica con RPi usando puerto serie por usb. RPi manda letreas codificadas en utf-8 y el Arduino PID los lee.

```
if (Serial.available() > 0)
   {
     switch (string)
```

Para dar comandos de los ángulos, RPi manda la letra “Z” (corresponde al número 90) lo cual resulta en el reseteo del contadorPi y la asignación del reseteo al pointer. Después RPi cuenta cada “1” (corresponde al número 49), cada vez aumentando el contador hasta recibir la letra de terminación “z.” Al recibir la señal de terminación, el valor de contador se transfiere al pointer y la marca que corresponde al servo particular se cambia a 1. A continuacón se observa el código para controlar el ángulo del servo abajo. La estructura para cambiar el ángulo encima es similar con una diferencia de letras mandadas por Pi.

```
    case 90:
    {
      contadorPi = 0;
      *contadorPiPointer = contadorPi;
      int string2 = Serial.read();
      while (string2 != 122) // Mientras que el pi no manda la letras de terminación (z), el Arduino continua de contar
      {
        string2 = Serial.read();
        if (string2 == 49) // Al recibir el valor "1" del Pi, el contador aumenta
        {
          contadorPi++;
        }
        if (string2 == 122) // Al recibir el señal de terminación (z), el contador transfiere su valor al variable global
         {
          *contadorPiPointer = contadorPi;
          MarcaControlRemoto = 1; // El modo de control remoto enciende para transferir el variable global del angulo mandado al Pi Servo.
          MarcaCambiarAngAbajo = 1;
        }
       }
     }
```

En Modo Ahorro se asigna el valor del pointer que llega de la sección anterior a la variable anguloDeseado y se llama una función que usa este valor.

```
void ModoAhorro()
{
  // APAGAR PROGRAMAS ANTERIORES-------------------------------------------------------------
  myPID.SetMode(MANUAL); // Apagar PID
  Ventilador(255); // Asegurar que el ventilador apague

  // MANDAR UN PULSO DE CONTROL AL SERVO-----------------------------------------------------
  if (MarcaControlRemoto == 1)
  {
    int anguloDeseado = * contadorPiPointer; // Obtener el valor mandado por Pi asignando el valor del pointer a un variable local
    ControlRemoto(anguloDeseado); // Llamar al programa que puede mandar el angulo deseado de Pi al Arduino Servos
  }
}
```

Dentro de *ControlRemoto()*, el Arduino PID recibe el ángulo a mandar y lo manda por el pin correspondiente según el comando de RPi. Al recibir el mensaje de RPi, dependiendo de la letra, MarcaCambiarAnguloAbajo o MarcaCambiarAnguloEncima cambia a 1 y el ángulo deseado se codifica en un pulso entre 800 y 1000ms. Usando *millis()* se pone la salida del pin correspondiente a la marca en HIGH por el tiempo correspondiente calculado.

```
void ControlRemoto(int anguloDeseado)
{
  delay(1000); // Asegurar que el señal mandado no tiene ruido de otras partes del Arduino

  long int pulso_high =  map(anguloDeseado, 0, 180, 800, 1000); // Hacer map del angulo para obtener la anchura de pulso

  MarcaControlRemoto = 1;

  long tref_pulso = millis(); // Mandar el pulso HIGH por el tiempo predeterminado en el paso anterior
  while(millis()-tref_pulso <= pulso_high)
  {
    if (MarcaCambiarAngAbajo == 1)
    {
      digitalWrite(pinControlServoAb, HIGH);
    }
    else
     {
       digitalWrite(pinControlServoEn, HIGH);

     }
  }
```

Después de mandar un pulso por el pin correspondiente, se manda un pulso LOW usando la misma estructura. Al final, se hace un reset de las marcas usadas.

```
  long int pulso_low = 1000-pulso_high;

  long tref_pulso2 = millis(); // Mandar el pulso LOW por el tiempo predeterminado en el paso anterior
  while(millis()-tref_pulso2 <= pulso_low)
  {
     if (MarcaCambiarAngAbajo == 1)
    {
      digitalWrite(pinControlServoAb, LOW);
    }
    else
     {
       digitalWrite(pinControlServoEn, LOW);
     }
  }
  tref_pulso2 = millis();

  MarcaControlRemoto=0; // Hacer reset del MarcadorControlRemoto para esperar mas comandos del Pi
  MarcaCambiarAngAbajo = 0;
  MarcaCambiarAngEncima = 0;

}
```

El Arduino Servo recibe los pulsos en el pin 7 o 12.

```
//VARIABLES-------------------------------PI MODO REMOTO
#define pinControlServoAb 7 // Pin que recibe pulsos del pi que decide el angulo a que mover
#define pinControlServoEn 12 // Pin que recibe pulsos del pi que decide el angulo a que mover
```

Se inicializa los pointers que guardan el valor recibido del ángulo.

```
int finalAnguloPi = 0; // Angulo de servo abajo por defecto cuando está en modo Pi
int *finAngPointer; // El pointer que refiere al angulo mandado de Arduino PID (que originalmente viene del Pi)

int finalAnguloPi2 = 5; // Angulo de servo arriba por defecto cuando está en modo Pi
int *finAngPointer2; // El pointer que refiere al angulo mandado de Arduino PID (que originalmente viene del Pi)
```

Durante el setup se indica que los pins que reciben los pulsos son INPUTs. Si el pinControlServoAb recibe el pulso, ejecute *ServoPiAbajo()*, mientras que pinControlServoEn ejecuta *ServoPiEncima()*.

```
pinMode(pinControlServoAb, INPUT); // El pulso que controla el Servo Abajo es un input
pinMode(pinControlServoEn, INPUT); // El pulso que controla es Servo Encima es un input
```

Se termina de inicializar los pointers previamente creados.

```
finAngPointer = &finalAnguloPi; // Setup del pointer del angulo abajo mandado de Arduino PID
finAngPointer2 = &finalAnguloPi2; // Setup del pointer del angulo encima mandado de Arduino PID
```

Si la MarcaPi = 1, el Arduino Servo vigila los pins previamente indicados y al recibir un pulso empieza a contar. Se usa *map()* para decodificar el pulso al ángulo. Se guarda el ángulo usando el pointer según el servo a mover. Abajo, se demuestra *ServoPiAbajo()* que tiene la misma estructura que *ServoPiEncima()*.

```
float anchoPulso;
float anguloPiAb;
int anguloPrevioAb; // Siempre empezar con el angulo 90 de servo abajo


void ServoPiAbajo()
{
  long int tref_pulso = millis();
  while(digitalRead(pinControlServoAb) == HIGH) // Recibir el puslo del Arduino PID
  {
    long int tiempo_corriente = millis();
    if(tiempo_corriente-tref_pulso < 1000 && tiempo_corriente-tref_pulso > 800) // Ignorar ruido que puede venir de otro Arduino
    {
     anchoPulso = tiempo_corriente-tref_pulso;
    }
    anguloPiAb = map(anchoPulso, 800, 1000, 0, 180); // Hacer map para obtener el angulo
    *finAngPointer = anguloPiAb;
  }
```

Se usa *read()* para obtener la posición más reciente del servo y se compara con el anguloPiAb, que era el comando original enviado por el Arduino PID. Se usa una estructura parecida a *ServoAutomatico()* para mover el servo un paso a la vez hasta que el ángulo mandado es igual a la posición más reciente del servo. Con este método incremental se obtiene más suavidad y menos backlash.

```
  anguloPrevioAb = bottom.read(); // Leer el angulo previo del servo para monitorizar/debug

  if (anguloPrevioAb > finalAnguloPi) // Comparar el valor del angulo que está asociado
                                      // con el pointer con el valor previo del servo abajo
   {
     servob = --servob; // Mover el motor un unido hasta en la dirección de arriba
                        // solo hasta llegar a su límite
     if (servob > servotLimitHigh)
     {
       servob = servotLimitHigh; // No hacer nada, porque ya está en el limite
     }

     //Serial.println(anguloPrevioAb); // Mover 1 unidad
     bottom.write(servob);
     anguloPrevioAb = bottom.read();
   }
 else if (anguloPrevioAb < finalAnguloPi)  // Comparar el valor del angulo que está asociado
                                           // con el pointer con el valor previo del servo abajo
   {
     servob = ++servob; // Mover el motor un unido hasta en la dirección de abajo
                        // solo hasta llegar a su límite
     if (servob < servotLimitLow)
     {
       servob = servotLimitLow; // No hacer nada, porque ya está en el limite
     }
     // Serial.println(anguloPrevioAb);
```

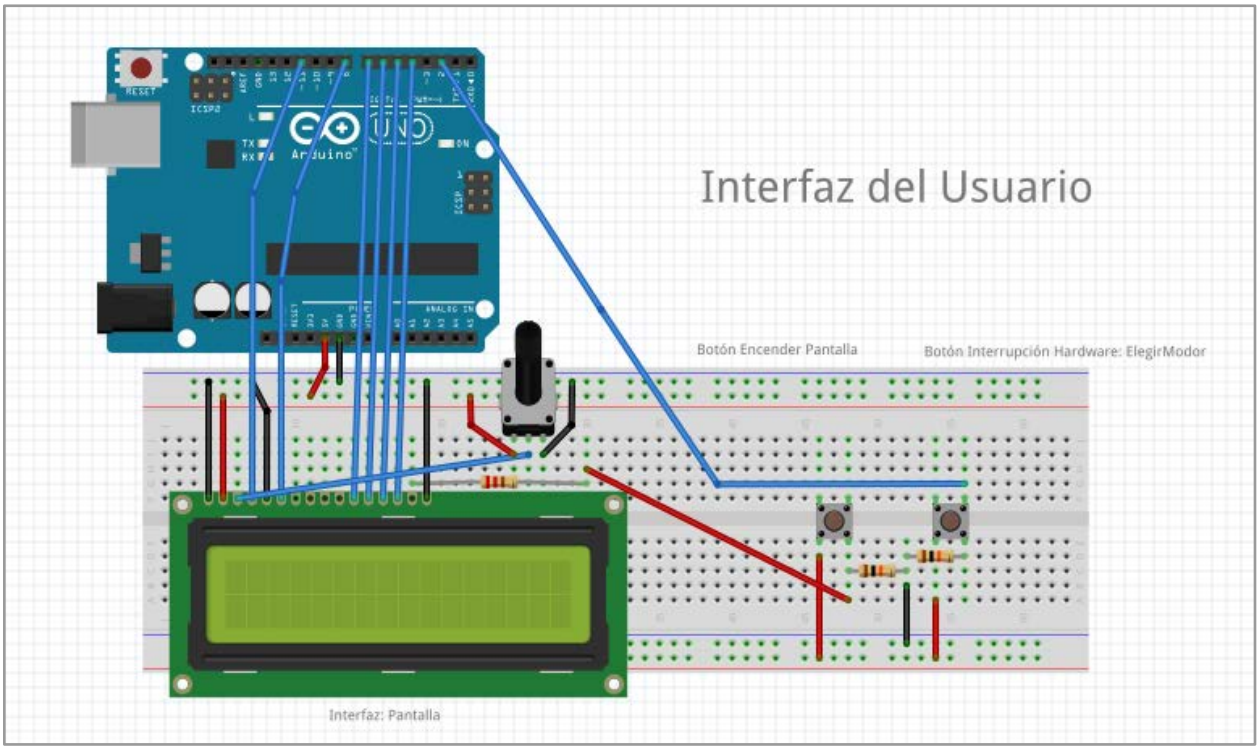
1.3. INTERFAZ

1.3.1. PANTALLA Y BOTONES

1.3.1.1. UTILIZACIÓN

Es importante tener alguna manera que permite a un usuario comunicarse con la máquina. Se incluye dos distintos modos que se pueden alternar con el “Botón Elegir Modo” y una manera que enciende la pantalla con el “Botón Encender Pantalla.”

1.3.1.2. MONTAJE



1.3.1.3. CÓDIGO

PANTALLA

Se inicializa las variables antes del bucle principal:

```
// VARIABLES-------------------------------PANTALLA
long int  tref_pantalla = millis();
const int maxtime = 10000; // Para no tener parpadeo en la pantalla, se usa millis()
```

Dentro de *loop()* se llama a *Pantalla()* con la frecuencia de refresco determinada por maxtime (arriba).

```
void loop()
{
// CONTROLA FRECUENCIA DE REFRESCO DE PANTALLA-----------------------------------------

  if (millis() - tref_pantalla >= maxtime) // Para no tener parpadeo en la pantalla, se usa Millis()
  {
    Pantalla(Temperatura(), Cargador());
    tref_pantalla = millis();
  }
```

Dentro de Pantalla() se usa setCursor para indicar la ubicación de los datos a escribir, rellenando la pantalla con información disponible según el modo (Ahorro o Normal) elegido.

```
#include <LiquidCrystal.h>

void Pantalla(int Temperatura, float porcentCargador)
{
  // INICIALIZAR PANTALLA-----------------------------------------------------------------
  LiquidCrystal lcd(12, 8, 7, 6, 4, 3); // Asignación de pins digitales a LCD
  lcd.begin(16,2); // Numero de columnas y filas
  lcd.clear(); // Asegurar clear antes de imprimir

  lcd.setCursor(0,0); // Donde empezar
  lcd.print("Modo:" ); // Que imprimir
  lcd.setCursor(6,0);
  lcd.print(MarcaModo);

  lcd.setCursor(0,1);
  lcd.print("Bateria" );

  if (porcentCargador < 100  && porcentCargador > 0) // Asegurar que la pantalla no imprime valores causados por el ruido
  {
   lcd.setCursor(8,1);
   lcd.print(porcentCargador); // Se el valor dado por Caragdor() no es causado por el ruido, imprimir el valor
  }
  else
  {
   lcd.setCursor(9,1);
   lcd.print("Espera"); // Imprimir esperar si el valor dado por Caragdor() es causado por el ruido
  }
```

En Modo Normal, el PID funciona, así que se puede observar la temperatura y notar su cambia gracias al funcionamiento del ventilador.

```
  // MOSTRAR DATOS QUE DEPENDEN DEL MODO-------------------------------------------------
  if (MarcaModo == 1)
  {
    lcd.setCursor(9,0);
    lcd.print("Temp:" );
    lcd.setCursor(14,0);
    lcd.print(Input);
  }
}
```

BOTONES

Una interrupción asociada al Botón M cambia el modo general (Ahorro-Normal).

```
// VARIABLES-------------------------------DE MODOS
#define BotonElegir = 2; //Botón que se usa para elegir el modo de funcionamiento
int MarcaModo = 0; //0=Modo Ahorro; 1=Modo Normal
```

Dentro del *setup()* se inicializa la interrupción que llama a *ElegirModo()*.

```
// SETUP INTERRUPCION de ELEGIR MODO-------------------------------------
pinMode(BotonElegir,INPUT_PULLUP);
attachInterrupt(0, ElegirModo,RISING); //añadir interrupción a pin 2
```

ElegirModo() es un software debounce que cuenta 100 ms al activar el botón M antes de cambiar el modo, lo cual evita una falsa activación del modo debido a ruido. También se puede usar un debounce de hardware con un condensador pequeño.

```
long int last_interrupt_time = millis();

void ElegirModo()
{
  // Si la interrupción es más rapida que 100ms, se considera como el bounce y lo se ignora
  if (millis() - last_interrupt_time > 100)
  {
    if (MarcaModo == 1) // Se usa marcas para indicar el modo (Ahorro o Normal)
    {
      MarcaModo = 0;
    }
    else
    {
      MarcaModo = 1;
    }
  }
  last_interrupt_time = millis();
}
```

Dependiendo del modo, el Arduino PID emite un pulso HIGH o LOW en el pinControModoArduino que está conectado al pinControlModoArduino de Arduino Servo lo cual resulta en una sincronización de dos Arduinos.

```
void loop()
{
  if (digitalRead(pinControlModoArduino) == HIGH) // El modo del Arduino Servos sólo puede estar cambiado por el pulso del Arduino PID
  {
    MarcaModo = 1; // El modo está cambiado con una marca
  }
  else
  {
    MarcaModo = 0;
  }
  //ELEGIR MODO-----------------------------------------------------------------------
  if (MarcaModo == 1)
  {
    ModoNormal();
  }
  else
  {
    ModoAhorro();
  }
}
```

El Arduino Servo usa la misma estructura para cambiar la MarcaPi con el botón C:

```
int MarcaPi = 1; //1 = Pi control ON, 0 = Pi control OFF
```

Dentro de *setup()* se conecta el pin de hardware interrupt a la función *ElegirModoPi()*

```
// SETUP INTERRUPCION de ELEGIR MODO-------------------------------------------------
attachInterrupt(digitalPinToInterrupt(3), ElegirModoPi, RISING); // Añadir interrupción a pin 3 que decide entre ServoManual y
                                                                 // ServoPi dentro de ModoAhorro.
                                                                 // El botón físico tiene prioridad.
```

Aunque he usado un hardware debounce (usando un condensador) en este caso, he incluido un software interrupt dentro del programa por si acaso hay un fallo con un condensador.

```
long int last_interrupt_time = millis();

void ElegirModoPi()
{
  if (millis() - last_interrupt_time > 0) // Si la interrupción es más rapida que 100ms, se considera como el bounce y lo se ignora
  {
    Serial.print("Elegir modo pi");
    if (MarcaPi == 1) // Se usa marcadores para indicar el modo.
    {
      MarcaPi = 0;
      Serial.println(MarcaPi);
    }
    else
    {
      MarcaPi = 1;
      Serial.println(MarcaPi);
    }
    last_interrupt_time = millis();
  }
}
```

Para encender y apagar la pantalla, he probado con tres métodos para encender y apagarla para decidir cuál es mejor para la estructura del proyecto.

El primer método usa el botón asociado a la pantalla como una interrupción. Al principio, he usado la estructura de *attachInterrupt(0, Pantalla, RISING)* para llamar la función que enciende la pantalla. Dependiendo del modo elegido, la pantalla imprimiría los valores correspondientes. Pero la interrupción no puede llamar a una función que requiere variables a ejecutar. En mi caso, la función *Pantalla()* requiere Temperatura y PorcentCargador. Para evitar ese problema

se inserta un transistor entre el cátodo de la pantalla y la tierra o se crea un pulsador que enciende la pantalla sólo al estar empujado. Por ya haber usado la interrupción con hardware en el caso de *ElegirModo(),* decidí explorar este método puramente de hardware que no depende del código. Por eso, he puesto el pulsador entre la parte que da alimentación a la pantalla y una resistencia conectada a tierra. Así he simplificado el código, añadido un segundo método de usar los pulsadores con un método de hardware puro, y he conseguido que la pantalla ahorre más energía al no estar siempre encendida. Al no depender del código, el Arduino no gasta la energía siempre vigilando un pin.

1.4. CARGADOR DE BATERIA

1.4.1.1. UTILIZACIÓN

Se utiliza un panel solar de baja potencia para cargar una batería AAA de NiMH. Aunque las baterías de Li-Ion son más populares para el uso con cargadores solares, tengo más dispositivos que usan baterías AAA.

1.4.1.2. COMPONENTES IMPORTANTES

Batería

En este proyecto, se usa una celda secundaria (celda recargable) AAA de 1Ah, 1.2V de NiMH.

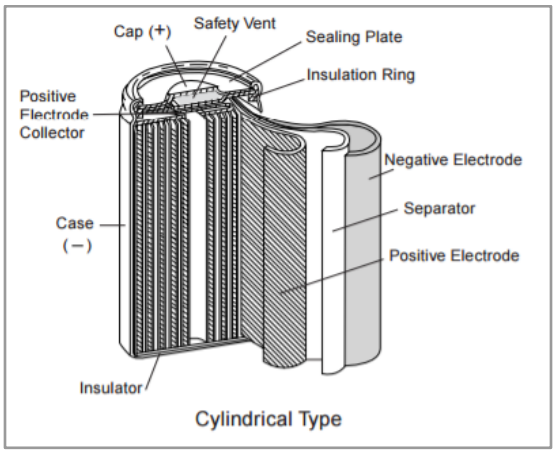


Figura 1.15: Representación de la estructura de la batería.

http://www2.emersonprocess.com/siteadmincenter

Una celda primaria cuya dirección de movimiento de los electrones entre ánodo y cátodo sólo es en un sentido y su reacción irreversible sólo puede ocurrir un número de veces predeterminado. En una celda secundaria ocurre una reacción reversible durante el proceso de recarga, resultando en una batería recargable hasta un número de recargas predeterminado según su tipo.

El ciclo de carga y descarga está basado en un proceso de oxidación y reducción reversible. Al conectar el ánodo y cátodo, ocurre una oxidación en que el ánodo pierde los electrones,

mientras que el cátodo se reduce y obtiene estos electrones. Se aplica energía, como en el proceso de recarga, para que los electrones puedan fluir en la dirección opuesta, restaurando el estado anterior de la batería. Se puede decir que al cambiar el proceso desde carga a descarga y al revés el ánodo y el cátodo se intercambian.

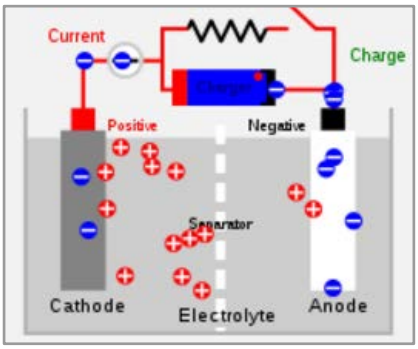


Figura 1.16: El proceso general de carga y descarga de la batería

https://es.wikipedia.org/wiki/Bater%C3%ADa_recargable

En el caso de batería NiMH, El $NiO(OH)$ es un ánodo mientras que una aleación de hidruro metálico es un cátodo.

Hay dos reacciones necesarias en transmisión de la energía (de la derecha a la izquierda, ocurre la descarga, mientras que al revés ocurre la recarga.):

1. Una reacción en el electrodo negativo

$$H_2O + M + e^- \leftrightharpoons OH^- + MH$$

En el electrodo positivo, una aleación de hidruro metálico (M) puede absorber o liberar los átomos de hidrógeno, dependiendo de la dirección. En el caso de este tipo de batería, este metal puede ser un metal de transición como titanio o zirconio (son parte del grupo AB_2) o puede ser un metal de tierra raro como lantano o cerio (son parte del grupo AB_5). Durante la recarga, H forma un enlace químico con M tras la electrólisis del agua (H y O_2 se separan cuando una corriente pasa por H_2O, idealmente a 1.23 V). Durante la descarga, el proceso ocurre al revés - MH y OH- forman agua, transformando M a su estado original y liberando un electrón.

2. Una reacción en el electrodo positivo

$$Ni(OH)_2 + OH^- \leftrightharpoons NiO(OH) + H_2O + e^-$$

Níquel (Ni) es un participante clave en la reacción del electrodo positivo que ofrece propiedades como baja corrosividad y alta durabilidad en el proceso. Se observa que $Ni(OH)_2$ y $NiO(OH)$ son reversibles dependiendo de la dirección del proceso. Durante la recarga, la energía externa convierte $Ni(OH)_2$ de nivel de energía bajo a su versión de alta energía $NiO(OH)$ mientras que un electrón está liberado. Durante la descarga, se convierte la molécula de Ni a su estado de baja energía, liberando energía, mientras absorbe un electrón.

La reacción general resultante del sistema es:

$$Ni(OH)_2 + M \leftrightharpoons NiO(OH) + MH$$

Se observa que la reacción general se basa en la transferencia del hidrógeno entre la molécula de Ni y M dependiendo de la dirección del proceso.

Si ocurre una sobrecarga, o si las condiciones no son favorables, la presión dentro de la pila empieza a aumentar peligrosamente debido a la producción de oxígeno excesivo por el electrodo positivo.

$$4OH^- \rightarrow O_2 + 2H_2O + 4e^-$$

Por eso, la capacidad del electrodo negativo es siempre mayor que la del electrodo positivo. Así, esta producción excesiva no ocurre sino que el oxígeno del electrodo positivo se difundirá hacia el electrodo negativo y se reconsumirá en las siguientes reacciones simultáneas:

$$4MH + O_2 \rightarrow 4M + 2H_2O$$

$$O_2 + 2H_2O + 4e^- \rightarrow 4OH^-$$

Afortunadamente, las pilas NiMH ofrecen ventajas numerosas comparando con otras baterías secundarias. Ofrecen 30-40 % más de capacidad (Ah) que las pilas NiCd comunes, pueden soportar un alto rango de temperatura (que es muy importante para un sistema solar que siempre estará expuesto a las condiciones medioambientales), y no contiene toxinas. Debido a la alta toxicidad del cadmio, es muy difícil reciclar o disponer de ellos, a diferencia de las pilas NiMH, que tienen una alta capacidad de ser reintegradas y recicladas.

Desafortunadamente, las celdas NiMH tienen los niveles de autodescargas más altos de su clase - un 20% de descarga durante las primeras 24 horas después de recarga y 10% cada mes después. Adicionalmente, se necesita un algoritmo avanzado para cargarlas segura y eficientemente ya que no responden bien a una sobrecarga a no ser que se use pequeñas corrientes.

En este proyecto, se carga la celda vigilando su voltaje actual junto con el temporizador que desconecta el transistor al alcanzar un tiempo predeterminado. *El voltaje se usa simbólicamente* para demostrar el nivel de carga ya que el nivel de voltaje en realidad depende de la rapidez de la carga y temperatura.

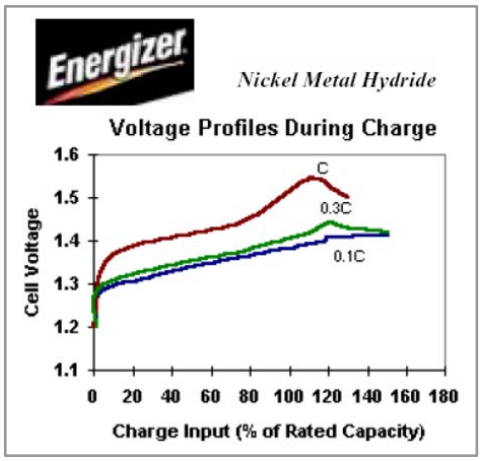


Figura 1.17: Se observa que debido a la corriente de carga y temperatura se puede alcanzar voltajes máximos distintos.

https://electronics.stackexchange.com/questions/155762/nimh-battery-excedes-charge-limit-when-plugged-into-tenergy-charger

Aunque la sobrecarga puede presentar peligro, el panel solar sólo puede proporcionar 0.125 A o en otras palabras ~C/10 de batería donde C, la capacidad en Ah de la batería, es 1Ah. Esto coloca al cargador en la categoría de “Cargadores Lentos” reduciendo el peligro significativamente porque el oxígeno producido tiene tiempo para ser reabsorbido en la reacción anterior. Junto con el temporizador, se crea un cargador simple.

Dentro de su clase, hay distintos tipos de baterías todos con sus ventajas y desventajas. Aunque existen baterías que son mucho menos sensibles a la sobrecarga y sufren menos autodescarga, la baja toxicidad, el buen precio, y alta Ah por peso se elige este tipo de pila para este proyecto.

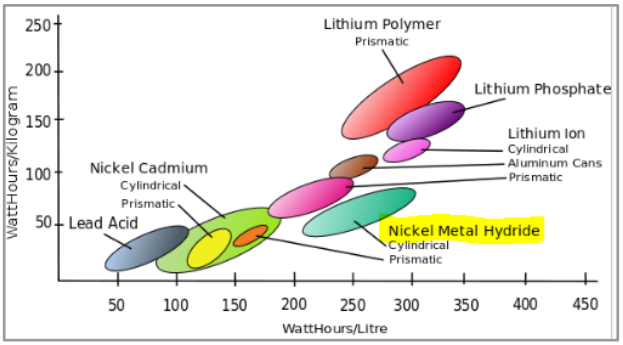


Figura 1.18: Comparación de Ah por peso de las pilas más comunes

https://es.wikipedia.org/wiki/Bater%C3%ADa_recargable

Panel Solar

El silicio es tetravalente en su estado natural y tiene espacio en su capa de valencia para 8 electrones. Las interacciones entre los átomos de silicio dejan una de carga neutral mientras satisfacen su necesidad de poseer 4 electrones más. Pero el dopaje de Fósforo, que tiene 5 electrones, resulta en un electrón extra lo cual significa que el panel (tipo N) tiene una carga negativa neta. Por otro lado, el dopaje de Boro, que tiene 3 electrones, resulta en un hueco extra lo cual significa que panel (tipo P) tiene una carga positiva neta. Se conecta ambos tipos con un hilo conductor que dirige el flujo de electrones que resulta cuando los fotones de sol liberan electrones del Fósforo que migran hasta los huecos de tipo P.

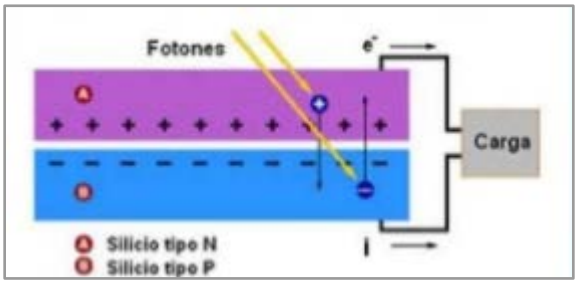


Figura 1.19: Funcionamiento del panel solar.

https://curiosoando.com/como-funciona-un-panel-solar

En este caso, se usa el panel solar de silicio cristalino de 12V, 1.5 W.

Figura 1.20: Foto del panel solar usado en el proyecto.

Así, con la información de la batería, se puede calcular el tiempo de carga:

$$Tiempo\ de\ Carga = \frac{Ah\ de\ batería}{\frac{Vatios\ de\ Panel}{Voltios\ de\ Panel}} = 8\ horas$$

No es necesario el regulador de carga en este caso porque la potencia del sistema es muy baja. Se puede determinar la necesidad del regulador de carga comparando el ratio de Ah máximo y los Amperios máximos del panel solar con el valor 200:

$\frac{1Ah}{0.125\ Amp}$? $200 \rightarrow 8 << 200 \Rightarrow$ no es necesario usar controlador de carga

En lugar de un algoritmo de control, se incorpora un temporizador y el método de “Carga Lenta” juntos con un voltímetro del Arduino.

Diodo Schottky

Se usa un diodo para prevenir el flujo de la corriente de la batería al panel solar durante la ausencia de la luz.

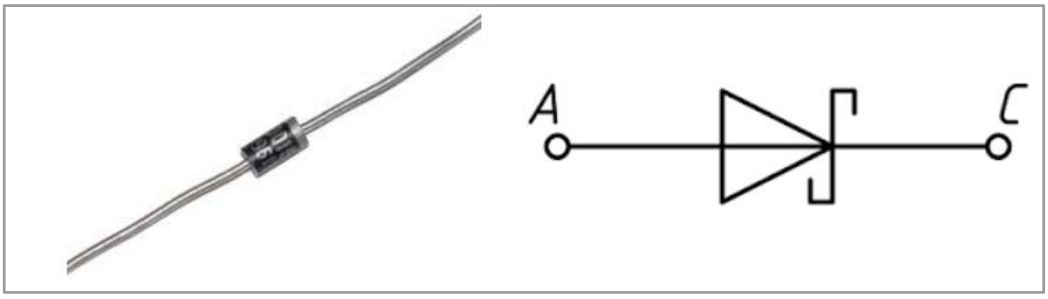


Figura 1.21: Diodo Schottky y su representación circuital.

http://www.vallecompras.com

Un diodo común de silicio es un semiconductor que consiste en una parte dopada de tipo P y otra de tipo N. El tipo P contiene más huecos que electrones que pueden llenarlos resultando en el movimiento de los electrones de un hueco a otro, lo cual resulta en el “movimiento” neto de los huecos. El tipo N tiene más electrones que huecos, lo cual resulta en el movimiento de los electrones. Al juntar ambas partes, los electrones excesivos de tipo N se mueven para llenar los huecos excesivos de tipo P resultando en una zona de depleción. Al aplicar por lo menos 0.7 V, la zona de depleción disminuye lo suficiente para dejar circular a la corriente.

El diodo Schottky está construido por unión metal-semiconductor en vez de la unión semiconductor tipo N y P. Se puede decir que, debido a esta estructura, el diodo Schottky es un “portador mayoritario” lo cual significa que sólo los electrones de tipo N afectan a su operación resultando en la ausencia de una recombinación, aumentando su rapidez y eficiencia. El diodo Schottky tiene una caída de voltaje en directo de 0.2 V en vez de 0.7 V de un diodo normal, lo que resulta en menos calentamiento y menos potencia gastada.

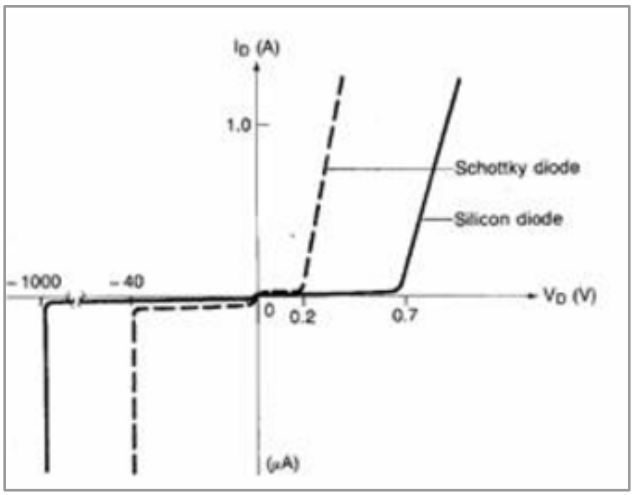
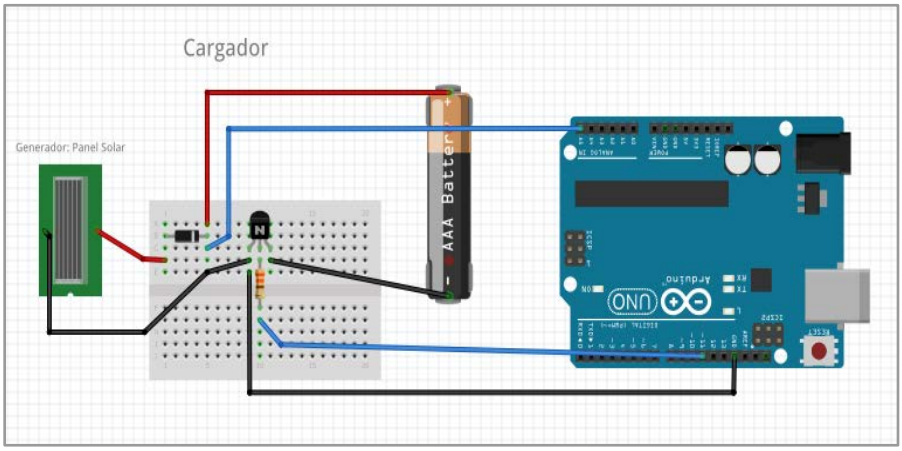


Figura 1.22: Diferencia entre comportamiento del diodo de silicio y diodo Schottky

http://www.monografias.com/trabajos-pdf2/diodo-schottky-barrera

1.4.1.3. MONTAJE



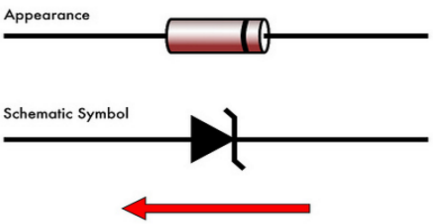


Figura 1.23: Orientación del diodo

http://www.evilmadscientist.com/2012/basics-introduction-to-zener-diodes/

Se conecta el panel solar con el diodo Shottky según la orientación del dibujo y después se conecta al terminal positivo de la batería. El terminal negativo se conecta al conector del transistor NPN. La base se conecta al Arduino a través de una resistencia de 330 ohmios. El emisor se conecta a la tierra compartida con el Arduino y el panel solar.

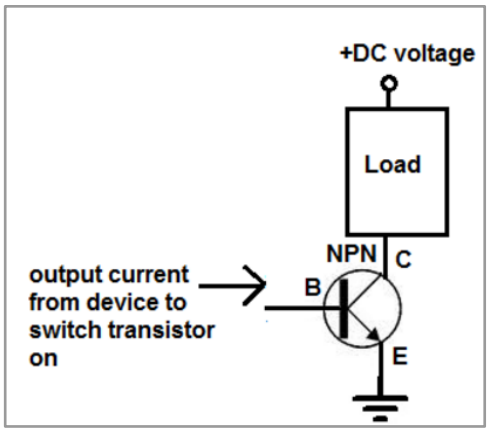
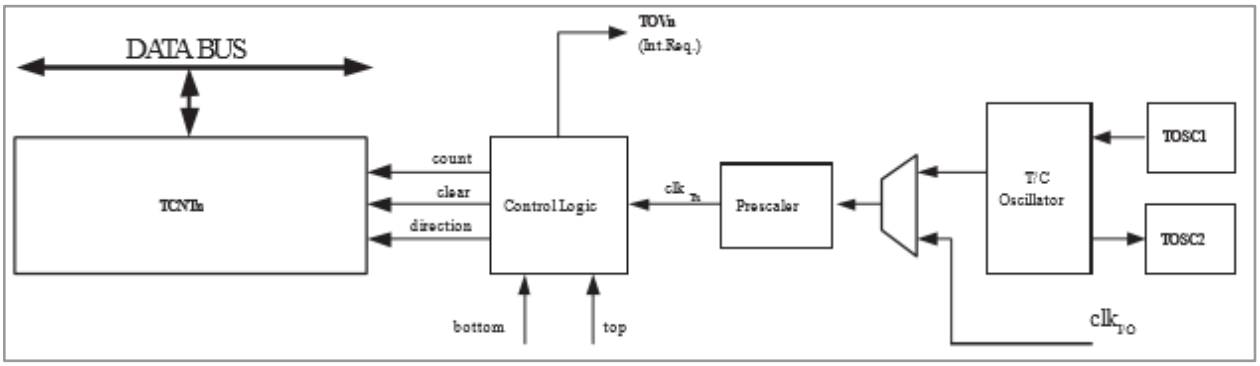


Figura 1.24: Orientación del transistor NPN

https://learn.sparkfun.com/tutorials/transistors/applications-i-switches

1.4.1.4. ISR de TEMPORIZADOR

Para no gastar energía esperando a la carga completa, se usa una interrupción de temporizador. El Arduino tiene tres temporizadores, los cuales son T0 de 8 bit (256 valores posibles), T1 de 16 bit (65536 valores posibles) y T2 de 8 bit (256 valores posibles). T1 está reservado para los servos, T0 está reservado para funciones como *delay()*, *millis()*, y *micros(),* mientras que T2 se puede usar para crear la ISR. Para aprender más sobre estas interrupciones, se modificó la librería de *MsTimer2* y se customizó el valor del incremento de cada ciclo. Generalmente, se puede crear ISR cambiando los bits específicos asociados con los bytes especiales.



En esta interrupción, la fuente del tiempo es el I/O clock, cuyos datos entran en el prescaler de 1024. A continuación, el Control Logic manda la dirección de contaje y el momento del ciclo del contaje. Se usa TCNT2 para definir el momento en el que occure el overflow y, tras múltiples overflows consecutivos, se llama al programa *Cargador()*.

Existen varios registros involucrados en el funcionamiento de T2:

TCCR2A

Bit	7	6	5	4	3	2	1	0
	COM2A1	COM2A0	COM2B1	COM2B0			WGM21	WGM20
Access	R/W	R/W	R/W	R/W			R/W	R/W
Reset	0	0	0	0			0	0

Los bits 0 y 1 controlan la secuencia de conteo, es decir, el tipo de máximo usado y el tipo de waveform producido. Se usa modo Normal, indicando que al contar, los valores del temporizador aumentan. 0xFF dentro de TOP y MAX dentro de TOV Flag Set se refiere a un valor máximo de la secuencia del conteo. La opción “Immediate” dentro de ”Update” de OCR0x deshabilita el “Double Buffer,” lo cual deshabilita la prevención de pulsos irregulares y da acceso a OCR2x directamente.

Durante el *setup(),* se modifica y se establece:

WGM21 ⇐ 0

WGM20 ⇐ 0

Mode	WGM22	WGM21	WGM20	Timer/Counter Mode of Operation	TOP	Update of OCR0x at	TOV Flag Set on[1]
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	-	-	-
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	-	-	-
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

TCCR2B

Bit	7	6	5	4	3	2	1	0
	FOC2A	FOC2B			WGM22	CS2[2:0]		
Access	R/W	R/W			R/W	R/W	R/W	R/W
Reset	0	0			0	0	0	0

Los bits CS2 dependen del prescaler elegido, que es 1024 en este caso. El prescaler deja modificar la cuenta del temporizador, trabajando con la frecuencia de 16 MHz del microprocesador. Si se usa el temporizador 1 (16 bits), se puede usar el prescaler para obtener una interrupción más lenta que la que puede producir un temporizador de 8 bits como T2. Por

lo tanto, se cuenta el número de overflows para extender la cuenta y tener flexibilidad al establecer.

CA22	CA21	CS20	Description
0	1	1	$clk_{I/O}$/32 (From prescaler)
1	0	0	clkI/O/64 (From prescaler)
1	0	1	$clk_{I/O}$/128 (From prescaler)
1	1	0	$clk_{I/O}$/256 (From prescaler)
1	1	1	$clk_{I/O}$/1024 (From prescaler)

Durante el *setup(),* para obtener el prescaler 1024, se modifica y se establece:

CS22 ⇐ 1

CS21 ⇐ 1

CS20 ⇐ 1

TIMSK2

Bit	7	6	5	4	3	2	1	0
						OCIEB	OCIEA	TOIE
Access						R/W	R/W	R/W
Reset						0	0	0

Este byte funciona con el byte TIFR2 y establece el uso del vector COMPA con el bit 1. Al principio, durante el *setup()* se deshabilita el vector compare:

OCIE2A ⇐ 0

Durante *ISRstart()*, se habilita el vector compare, que nos deja implementar la interrupción tras un número determinado de overflows.

ASSR

Bit	7	6	5	4	3	2	1	0
		EXCLK	AS2	TCN2UB	OCR2AUB	OCR2BUB	TCR2AUB	TCR2BUB
Access		R	R	R	R	R	R	R
Reset		0	0	0	0	0	0	0

EXCLK apaga el oscilador de cristal y se elige el reloj asíncrono mientras que AS2 permite usar al I/O reloj como la fuente de reloj en lugar del oscilador de cristal. Así, durante el *setup()*, se modifica:

AS2 ⇐ 0

EXCLK ⇐ 1

TCNT2

Es el incremento de tiempo de cada ciclo antes de que ocurra el overflow. He elegido un número 752, lo cual se traduce en aproximadamente 1ms. Con la rapidez predeterminada, el microprocesador sale y entra del *ISR(TIMER2_OVF_vect)* contando el número de veces que se entra y comparándolo con el valor *incrementoLlamada,* que equivale a 1000 (1 milisegundo). Si se supera este valor, se llama a la función Cargador().

Durante el *setup()*, se define este valor, mientras que en *ISRstart()* se le asigna a TCNT2.

1.4.1.4. CÓDIGO

Antes del *setup()* se conecta un pin analógico a un cable que mide la tensión de la batería y otro a un pin digital que puede conectar y desconectar el transistor para abrir el circuito y interrumpir la carga. Se declara la variable que almacena el porcentaje de la carga y se inicializa la marca de carga (ésta se utiliza sólo dentro de la comunicación con el RPi).

```
// VARIABLES----------------------------------CARGADOR
#define pinVoltaje A0
#define pinTransistor 11
int porcentCargador;
int MarcaCarga = 1;
```

Durante el setup(), se establece el pinVoltaje como INPUT, mientras que el pinTransistor es un OUTPUT.

```
// SETUP CARGADOR-----------------------------------------------------------
pinMode(pinVoltaje,INPUT);
pinMode(pinTransistor,OUTPUT);
```

También se declara las variables del ISR. Se usa “volatile long” para modificar los variables globales dentro de la interrupción.

```
// VARIABLES---------------------------------ISR TEMPORIZADOR CARGADOR
volatile int overflowing; // Marcador de overflow
volatile long count;
volatile long tcnt2;
unsigned long incrementoLlamada = 1000; // Tiempo que va a pasar entre llamadas de Cargador() dentro de ISR (en ms). 10000
volatile long countDisconnect = 0;
volatile long *countDisconnectPointer;
```

Se inicializa la interrupción del temporizador y se rellena los bits según la parte anterior.

```
// SETUP ISR---------------------------------------------------------------
ISRsetup();
ISRstart();
```

```
void ISRsetup()

// REINICIAR TODO AL PRINCIPIO---------------------------------------------------------------------------
{
  TIMSK2 &= ~(1<<TOIE2); // Deshabilitar overflow

  TCCR2A &= ~((1<<WGM21) | (1<<WGM20)); // Poner Modo de contar: Normal; TOP: 0xFF; update: immediate; TOV Flag Set: Max
  TCCR2B &= ~(1<<WGM22);

  ASSR &= ~(1<<AS2); // Usar reloj de I/O
  ASSR |= (1<<EXCLK);

  TIMSK2 &= ~(1<<OCIE2A); // Deshabilitar vector compare

  //PRESCALER (CPU 16 mHz)---------------------------------------------------------------------------
  TCCR2B |= (1<<CS22);
  TCCR2B |= ((1<<CS21) | (1<<CS20));

  tcnt2 = 752; // Con medidas, he encotrado que con este valor, la duración de cada contaje es 1ms. 1000 ciclos de contaje ~ 1 seg
}

void ISRstart()

// INICIALIZAR CONTADOR---------------------------------------------------------------------------
{
  count = 0;
  overflowing = 0;
  TCNT2 = tcnt2;
  TIMSK2 |= (1<<TOIE2);
}
```

He calculado tcnt2 para entrar en ISR y aumentar el count cada milisegundo. Cuando el count es igual o mayor al incrementoLlamada (1000 ms), se llama a Cargador() y se aumenta la variable countDisconnect, que después de cierto punto desconecta el cargador usando el transistor. Se calcula el valor máximo de countDisconnect usando el tiempo necesario para cargar la batería. Si es necesario 8 horas para cargar la batería y el countDisconnect aumenta cada 1000 ms = 1 seg, se puede decir que después de $8\ hrs\ *\ \frac{3600\ seg}{1\ hr} = 28800$ incrementos, se debe desconectar el cargador.

```
// ISR TEMPORIZADOR CARGADOR-----------------------------------------------------------------
ISR(TIMER2_OVF_vect)
{
  TCNT2 = tcnt2;
  count++; // Aumentar el count cada vez cuando ISR funciona

  if(count >= incrementoLlamada && !overflowing) // Si el count es igual a nuestro incremento de
                                                 // llamada predeterminada, llamar a función Cargador()
  {
    overflowing = 1; // Set el marcador de overflow
    countDisconnect++;
    *countDisconnectPointer = countDisconnect; // Este contador aumenta hasta que pasa el tiempo predeterminado
                                               // de cargar la batería. Después, el cargador se desconecta.
    count = count - incrementoLlamada;
    Cargador(); // Llamar el programa dentro de ISR
    overflowing = 0; // Reset el marcador de overflow
  }
```

Al ejecutar *Cargador()*, se crea una función similar a *map()* pero que puede usar floats y se calcula el porcentaje de carga según el voltaje de la batería.

```
// USAR MAP PARA OBTENER PORCENTAJE TE CARGA-------------------------------------------
//(x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
float voltajeCircuito = (voltajeBateriaAnalog - 0.0) * (5.0 - 0.0) / (1023.0 - 0) + 0;  // Hacer map con floats para obtener voltaje entre valor 0 y 5
float porcentCargador = (voltajeCircuito - 0.78) * (100 - 0.0) / (1.34 - 0.78) + 0;     // El voltaje de batería completamente descargada es 0.78 V,
                                                                                        // mientras que el voltaje de una batería completamente cargada es 1.34 V
```

Se usa el pointer *countDisconnectPointer para pasar el valor del contador a la variable countDisconnect dentro de *Cargador()* y se compara con 28800 (previamente calculado). Después de 8 horas o cuando el voltaje alcance 1.34 V el transistor desconecta la batería. La función devuelve el porcentaje de carga que se usa en la función *Pantalla()*.

```
if (voltajeCircuito > 1.34 || countDisconnect == 28800 || MarcaCarga == 0 )  // Cada 1000ms o 1 seg el countDisconnect incrementa.
{                                                                            // Cargar bateria toma 8 hrs o 28800 seg. => 28800 incrementos.
  digitalWrite(pinTransistor, LOW); // Apagar con transistor
}
else
{
  digitalWrite(pinTransistor, HIGH); // Encender con transistor
}

return porcentCargador; // Sirve para dar una aproximación al usuario de nivel de carga
```

2. MOVIMIENTO DE SERVOS

Al principio, era posible conectar los servos al Arduino sin ningún problema. Pero al expandir el código, los servos empezaron a moverse más lentamente. Para intentar resolver este problema he seguido los siguientes pasos:

1. Una fuente de alimentación fue proporcionada a los servos en un circuito separado con una masa común con el resto de circuito. Usando el polímetro, he asegurado que el voltaje de entrada era el prometido por la fuente. Para asegurar que los circuitos estaban verdaderamente separados, he intentado mover los servos sin enchufar la fuente de alimentación, lo cual he hecho con éxito, pero sin ninguna mejora del movimiento de los servos.

2. He desenchufado todo salvo un servo conectado a una fuente de alimentación distinta a la del Arduino, que está conectado por el USB al ordenador. Al mover los servos, no se nota la diferencia.

3. He enchufado otros servos al sistema, en caso de que los *actuales* estén quemados. Al mover los servos, no se nota la diferencia.

4. He creado un programa aparte que puede mover los servos usando el mismo código dentro de mi programa principal, pero sin ninguna otra función (como ventilador, pantalla, etc.). Así, se nota que los servos tenían un movimiento mejorado.

Se puede concluir que el problema inicial no era de corriente, ni del estado de los servos, sino que era el ruido debido a la cantidad de código. Con más código, y más piezas enchufadas, parece que hay menos suavidad del movimiento de los servos debido al ruido.

Para resolver el problema he añadido dos condensadores: uno de 2200µF en paralelo con la fuente de alimentación separado de los servos, y otro del mismo valor en paralelo con la fuente de alimentación del Arduino.

Después, se observa que hay una mejora del movimiento de los servos, aunque todavía hay una patada pequeña que ocurre al encender el ISR del temporizador. Se asegura de que los servos no usan T2 para controlar sus movimientos (usan T1) y de que el encendido ocurre sólo en *setup()* del programa. Al final, al cambiar cada *delay()* a *millis()*, el movimiento de los servos fue mejorando.

Al final, se puede concluir que reducir el ruido con condensadores y no usar funciones múltiples a la misma vez dentro un microprocesador tan pequeño como un Arduino Uno puede significativamente aumentar la claridad de la señal que sale del Arduino.

3. FUENTE DE ALIMENTACIÓN

En el caso de este proyecto, dónde es necesario alimentar dos servos y un ventilador, se calcula que la corriente necesaria puede ser entre 1 y 2 amperios según el tipo de movimiento y la carga sobre los servos. Debido a la necesidad de más corriente, una fuente de alimentación de USB fue construida al principio y soldada por separado. Con una placa metálica, no es necesario preocuparse de que la corriente sobrepase 1 amperio.

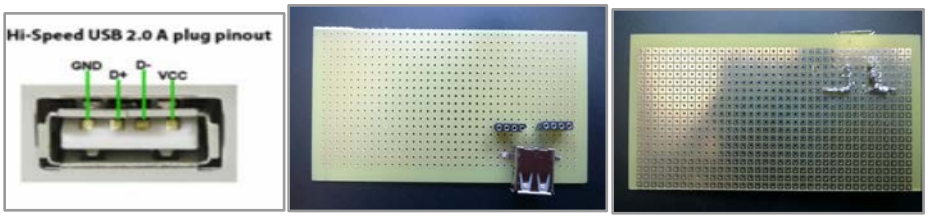


Figura 3.1: La base de la fuente de alimentación

https://www.quora.com/How-does-a-usb-cable-charge-a-device-as-well-as-transfer-data

Cada cable puede soportar 1 amp, lo cual significa que se pueden usar con los servos.

Con un polímetro se nota que con el USB junto a las conexiones soldados, el voltaje no era muy estable. Por eso he añadido dos opciones para alimentar a los servos:

1. El transformador buck fue soldado junto con una entrada jack.
2. He enchufado un micro USB adaptador para poder alimentar el circuito con un cargador de ~5V, 2A.

Usando el polímetro, se observa que la fuente era muy estable y que sí estaba dando el voltaje prometido a los servos. Aún con el uso de una fuente dedicada a los servos, no mejoró sus movimientos. Sin embargo, los condensadores cerca del USB pararon los movimientos espásticos de los servos durante su funcionamiento debido a los golpes de corriente causados por otros componentes del circuito. En otras palabras, la separación de los circuitos puede reducir el ruido.

4. ANEXO

4.1 CALIBRACIÓN

CARGADOR:

Se usa la información general de Energizer para obtener las curvas de carga para pilas de NiMH y aprender más sobre el proceso complejo y no lineal de carga y descarga de este tipo de baterías.

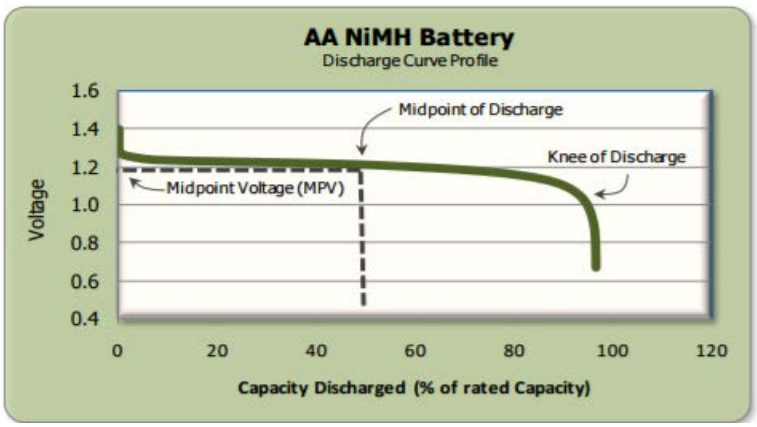


Figura 4.1: Comportamiento de batería NiMH durante la descarga

Es necesario subrayar que el voltaje máximo de la batería puede depende de condiciones como la temperatura y la corriente que la carga. No obstante, se usa este proceso para aprender más sobre la batería NiMH y para obtener los rangos generales para servir como una fuente de información para el usuario con el uso de pantalla.

Después de cargar la batería por separado y medir el voltaje máximo (1.29 V), se descarga la batería y se obtiene el voltaje mínimo (0.73 V). Con esto se puede calibrar, respectivamente, la carga y la descarga completa de la batería por lo cual se puede usar la función *map()* para obtener el porcentaje actual de la carga.

Después de montar la estructura, es necesario hacer más calibración, pero esta vez con el programa del Arduino. Se observa que cuando el Arduino mide 0.87, mi polímetro mide 0.82 Similarmente, cuando mi polímetro mide 1.29 V, el uno mide 1.34 V. Por eso, es necesario usar valores del Arduino, para evitar este error de 0.5V.

Es necesario mencionar que lo más importante, por razones de la seguridad, es el voltaje máximo, ya que una sobrecarga de una batería NiMh puede resultar en un peligro de fuego. Así se usa el valor máximo como el trigger que abre el transistor y rompe la conexión entre el panel solar y la batería, parando el proceso de carga.

SERVOS:

Se usa un programa que contiene sólo una función que mueve los servos desde 0 a 180 grados para determinar los puntos mínimos y máximos de los servos y de la estructura. Moviendo los servos a los extremos distintos se marca los ángulos correspondientes en la estructura y así se elige la posición inicial del seguidor. Se nota que la estructura limita el movimiento de los servos ya que el punto mínimo del servo encima es 5 grados y el máximo es 90, mientras que el punto mínimo del servo abajo es 0 grados y el punto máximo 180.

Al principio, se usó una estructura de 100 por ciento de cartón, pero se cambió a una de plástico para aumentar la robustez del seguidor, aunque todavía se observa una falta de tracción fuerte entre los servos y sus conexiones a la estructura.

4.2 CÓDIGO COMPLETO

ARDUINO PID

```
// LIBRERIAS INCLUIDAS Y USADAS----------
#include <PID_v1.h>
#include <dht11.h>


// VARIABLES-------------------------------------PID
#define tempPin 15   // Antes era 13
float temperatura = 0;
double consKp = 70, consKi=0.125, consKd=0; // Definir Tuning Parameters
double Input, Output, Setpoint;
PID myPID(&Input, &Output, &Setpoint, consKp, consKi, consKd, REVERSE);

// VARIABLES---------------------------------DE MODOS
const int BotonElegir = 2; // Botón que se usa para elegir el modo de funcionamiento
```

```
int MarcaModo = 0; // 0=Modo Ahorro; 1=Modo Normal
int MarcaControlRemoto = 1;  // Manda el Arduino PID esperar al angulo para mandar a Arduino Servo

// VARIABLES----------------------------------VENTILADOR
#define pinVentilador 5

// VARIABLES----------------------------------CARGADOR
#define pinVoltaje A0
#define pinTransistor 11
int porcentCargador;
int MarcaCarga = 1;

// VARIABLES----------------------------------PANTALLA
long int  tref_pantalla = millis();
const int maxtime = 10000; // Para no tener parpadeo en la pantalla, se usa millis()

// VARIABLES----------------------------------ISR TEMPORIZADOR CARGADOR
volatile int overflowing; // Marcador de overflow
volatile long count;
volatile long tcnt2;
unsigned long incrementoLlamada = 1000; // Tiempo que va a pasar entre llamadas de Cargador() dentro de ISR (en ms).
volatile long countDisconnect = 0;
volatile long *countDisconnectPointer;

// VARIABLES----------------------------------PI MODO REMOTO (se controla pin 6 de Arduino Servo con pin 9 de Arduino PID)
#define pinControlModoArduinoServo 9 // Cambia el modo de Arduino Servo
#define pinControlServoAb 10 // Manda pulso a Arduino Servo para cambiar angulo de Servo Abajo
#define pinControlServoEn 13 // Manda pulso a Arduino Servo para cambiar angulo de Servo Encima
int contadorPi = 0; // Utilizado en obtener comando de angulo de Pi
int *contadorPiPointer;
int MarcaCambiarAngEncima = 0; // Decide a que servo tiene que mover en Arduino Servo
int MarcaCambiarAngAbajo = 0;

void setup()
{
 // SETUP INTERRUPCION de ELEGIR MODO-----------------------------------
 pinMode(BotonElegir,INPUT_PULLUP);
 attachInterrupt(0, ElegirModo,RISING); // Añadir interrupción a pin 2

 // SETUP IMPRIMIR-------------------------------------------------------
 Serial.begin(9600);

 // SETUP CARGADOR-------------------------------------------------------
 pinMode(pinVoltaje,INPUT);
 pinMode(pinTransistor,OUTPUT);

 // SETUP PID------------------------------------------------------------
 pinMode(pinVentilador,OUTPUT); // Ventilador
 myPID.SetOutputLimits(0,255); // Crear los limites de PID
 myPID.SetMode(MANUAL); // No encender PID

 // SETUP ISR------------------------------------------------------------
 ISRsetup();
 ISRstart();

 // SETUP PI MODO REMOTO-------------------------------------------------
 pinMode(pinControlServoAb,OUTPUT); // Pin que manda pulsos al Arduino Servo
 pinMode(pinControlServoEn,OUTPUT); // Pin que manda pulsos al Arduino Servo

 pinMode(pinControlModoArduinoServo,OUTPUT); // Pin que manda pulsos al Arduino Servos

 contadorPiPointer = &contadorPi;
 countDisconnectPointer = &countDisconnect;
}
```

```
void loop()
{
// CONTROLA FRECUENCIA DE REFRESCO DE PANTALLA----------------------------------------

  if (millis() - tref_pantalla >= maxtime) // Para no tener parpadeo en la pantalla, se usa Millis()
  {
    Pantalla(Temperatura(), Cargador());
    tref_pantalla = millis();
  }

  // PI-------------------------------------------------------------------------------
 if (Serial.available() > 0)
    {
      int string = Serial.read();
      switch (string)
      {
        // CONTROL CON PI (MODOS)-----------------------------------------------------------------
        case 65: // Modo (A)horro
        {
          MarcaModo=0;
          Serial.println(MarcaModo);
          break;
        }
        case 78: // Modo (N)ormal
        {
          MarcaModo=1;
          Serial.println(MarcaModo);
          break;
        }
        case 73: // (I)nterrumpir proceso de cargar bateria
        {
          MarcaCarga=0;
          Serial.println(MarcaCarga);
          break;
        }

        // INFORMACION MANDADO A PI---------------------------------------------------------
        case 116:  // (t)emperatura
        {
          float message = Temperatura();
          Serial.println(message);
          break;
        }
        case 112: // (p)orcentaje de bateria
        {
          int message = Cargador();
          Serial.println(message);
          break;
        }
        case 109: // (m)odo corriente
        {
          Serial.println(MarcaModo);
          break;
        }
        case 101: // angulo de servo (e)ncima
        {
          int angulot = analogRead(A4);
          angulot = map(angulot, 0, 1023, 5, 90);
          Serial.println(angulot);
          break;
        }
        case 97: // angulo de servo (a)bajo
        {
          int angulob = analogRead(A5);
```

```
      angulob = map(angulob, 0, 1023, 0, 180);
      Serial.println(angulob);
      break;
     }
     case 90: // (Z) mandado de pi empieza el bucle de espera del angulo
     {
      contadorPi = 0;
      *contadorPiPointer = contadorPi;
      int string2 = Serial.read();
      while (string2 != 122) // Mientras que el pi no manda la letras de terminación (z o y), el Arduino continua de contar
      {
       string2 = Serial.read();
       if (string2 == 49) // Al recibir el valor "1" del Pi, el contador aumenta
       {
        contadorPi++;
       }
       if (string2 == 122) // Al recibir el señal de terminación (z), el contador transfiere su valor al variable global
        {
         *contadorPiPointer = contadorPi;
         MarcaControlRemoto = 1; // El modo de control remoto enciende para transferir el variable global del angulo
                                 // mandado al Pi Servo.
         MarcaCambiarAngAbajo = 1;
       }
      }
      break;
    }
     case 89:  // (Y) mandado de pi empieza el bucle de espera del angulo
     {
      contadorPi = 0;
      *contadorPiPointer = contadorPi;
      int string2 = Serial.read();

      while (string2 != 121) // Mientras que el pi no manda la letras de terminación  (y), el Arduino continua de contar
      {
       string2 = Serial.read();
       if (string2 == 49) // Al recibir el valor "1" del Pi, el contador aumenta
       {
        contadorPi++;
       }

       if (string2 == 121) // Al recibir el señal de terminación (y), el contador transfiere su valor al variable global
       {
        *contadorPiPointer = contadorPi;
        MarcaControlRemoto = 1; // El modo de control remoto enciende para transferir el variable global del angulo
                                // mandado al Pi Servo.
        MarcaCambiarAngEncima = 1;

       }
      }
      break;
     }
    }
  }


// ELEGIR MODO--------------------------------------------------------------------------
if (MarcaModo == 1) // Decide el modo de funcionamiento del Arduino PID y el Arduino Servos
{
 ModoNormal();
 digitalWrite(pinControlModoArduinoServo, HIGH);  // Manda el mensaje al pin del Arduino Servos para cambiar el modo
}
else
{
 ModoAhorro();
 digitalWrite(pinControlModoArduinoServo, LOW);
```

```
  }
 // Serial.println(MarcaModo);
}


 // ISR TEMPORIZADOR CARGADOR-----------------------------------------------------------
 ISR(TIMER2_OVF_vect)
 {
  TCNT2 = tcnt2;
  count++; // Aumentar el count cada vez cuando ISR funciona

  if(count >= incrementoLlamada && !overflowing) // Si el count es igual a nuestro incremento de
                                                 // llamada predeterminada, llamar a función Cargador()
  {
    overflowing = 1; // Set el marcador de overflow
    countDisconnect++;
    *countDisconnectPointer = countDisconnect; // Este contador aumenta hasta que pasa el tiempo predeterminado
                                               // de cargar la batería. Después, el cargador se desconecta.
    count = count - incrementoLlamada;
    Cargador(); // Llamar el programa dentro de ISR
    overflowing = 0; // Reset el marcador de overflow
  }
 }
```

```
float Cargador()
{
 // LEER PIN----------------------------------------------------------------------------
 delay(5);
 float voltajeBateriaAnalog = analogRead(pinVoltaje); // Leer entrada de pin A4. Se lo convierte en un valor de voltaje.

 // USAR MAP PARA OBTENER PORCENTAJE TE CARGA-------------------------------------------
 // (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
 float voltajeCircuito = (voltajeBateriaAnalog - 0.0) * (5.0 - 0.0) / (1023.0 - 0) + 0;  // Hacer map con floats para obtener
                                                                                         // voltaje entre valor 0 y 5
 float porcentCargador = (voltajeCircuito - 0.78) * (100 - 0.0) / (1.34 - 0.78) + 0;    // El voltaje de batería completamente
                                                                                         // descargada es 0.78 V,
                                                                                         // mientras que el voltaje de una batería
                                                                                         // completamente cargada es 1.34 V


 // DESCONECTAR BATERIA AL COMPLETAR CARGAR---------------------------------------------
 countDisconnect = *countDisconnectPointer; // Cada vez que Cargador() está ejecutado, el contador aumenta,
                                            // y su valor global pasa al variable local.

 Serial.print("Interrupcion Temporizador: Cargador----------");
 Serial.print("Voltaje Bateria");
 Serial.print("\t");
 Serial.println(voltajeCircuito);

 if (voltajeCircuito > 1.34 || countDisconnect == 28800 || MarcaCarga == 0 )  // Cada 1000ms o 1 seg el countDisconnect
                                                                              // incrementa.
 {                                                                            // Cargar bateria toma 8 hrs o 28800 seg. =>
                                                                              // 28800 incrementos.
  digitalWrite(pinTransistor, LOW); // Apagar con transistor
 }
 else
 {
  digitalWrite(pinTransistor, HIGH); // Encender con transistor
 }

 return porcentCargador; // Sirve para dar una aproximación al usuario de nivel de carga
}
```

```
long int last_interrupt_time = millis(); // Se guarda valores static sin destruirlas al salir de la función diferente a variables locales

void ElegirModo()
{
  // Si la interrupción es más rapida que 200ms, se considera como el bounce y lo se ignora. Usar 0 con hardware debounce
  if (millis() - last_interrupt_time > 0)
  {
    if (MarcaModo == 1) // Se usa marcadores para indicar el modo.
    {
      MarcaModo = 0;
    }
    else
    {
      MarcaModo = 1;
    }
  }
  last_interrupt_time = millis();
}
```

```
void ISRsetup()

// REINICIAR TODO AL PRINCIPIO----------------------------------------------------------------------
{
  TIMSK2 &= ~(1<<TOIE2); // Deshabilitar overflow

  TCCR2A &= ~((1<<WGM21) | (1<<WGM20)); // Poner Modo de contar: Normal; TOP: 0xFF; update: immediate; TOV Flag Set:
Max
  TCCR2B &= ~(1<<WGM22);

  ASSR &= ~(1<<AS2); // Usar reloj de I/O
  ASSR |= (1<<EXCLK);

  TIMSK2 &= ~(1<<OCIE2A); // Deshabilitar vector compare

  //PRESCALER (CPU 16 mHz)---------------------------------------------------------------------------
  TCCR2B |= (1<<CS22);
  TCCR2B |= ((1<<CS21) | (1<<CS20));

  tcnt2 = 752; // Con medidas, he encotrado que con este valor, la duración de cada contaje es 1ms. 1000 ciclos de contaje ~ 1
                  // seg
}

void ISRstart()

// INICIALIZAR CONTADOR-----------------------------------------------------------------------------
{
  count = 0;
  overflowing = 0;
  TCNT2 = tcnt2;
  TIMSK2 |= (1<<TOIE2);
}
```

```
void ModoAhorro()
{
  // APAGAR PROGRAMAS ANTERIORES-----------------------------------------------------------
  myPID.SetMode(MANUAL); // Apagar PID
```

```
  Ventilador(255); // Asegurar que el ventilador apague

  // MANDAR UN PULSO DE CONTROL AL SERVO-----------------------------------------------
  if (MarcaControlRemoto == 1)
  {
    int anguloDeseado = * contadorPiPointer; // Obtener el valor mandado por Pi asignando el valor del pointer a un variable
       // local
    ControlRemoto(anguloDeseado); // Llamar al programa que puede mandar el angulo deseado de Pi al Arduino Servos
  }
}
```

```
void ModoNormal()
{
 // LLAMAR PID()-----------------------------------------------------------------------
  myPID.SetMode(AUTOMATIC);

  Ventilador(pid());
}
```

```
long int tref_temperatura = millis();
long int tref_print; // Depurar

int pid()
{
  if (millis() - tref_temperatura >= 1000) // Para evitar el uso de delay, y dejarlo sólo cuando no hay otro opción
  {                                        // se usa millis() para reducir la frequencia de leída
    Input = Temperatura();  // Llamar Temperatura() que es el Input al PID
    tref_temperatura = millis();
  }

  Setpoint = 27; // Setpoint del PID
  myPID.Compute(); // Calcular output del PID
  int Speed = map(Output, 255, 0, 0, 190); // Dar el valor que el ventilador entiende. Usando Keyes Ventilador,
                                           // 0 PWM = Max velocidad y 190 = Parada

  if (millis() - tref_print >= 1000) // Para evitar el uso de delay, y dejarlo sólo cuando no hay otro opción, se usa millis() para
  // reducir la frequencia de leída
  {
  Serial.print("PID-----------------------------------------------------------------------------------------");
  Serial.print("Input=");
  Serial.print("\t");
  Serial.print(Input);
  Serial.print("\t");
  Serial.print("Output=  ");
  Serial.print("\t");
  Serial.println(Output);
  tref_print = millis();
  }
  return Speed; // Devolver Speed, lo que alimenta Ventilador()
}
```

```
#include <LiquidCrystal.h>

void Pantalla(int Temperatura, float porcentCargador)
{
  // INICIALIZAR PANTALLA-------------------------------------------------------------
  LiquidCrystal lcd(12, 8, 7, 6, 4, 3); // Asignación de pins digitales a LCD
```

```
  lcd.begin(16,2); // Numero de columnas y filas
  lcd.clear(); // Asegurar clear antes de imprimir

  lcd.setCursor(0,0); // Donde empezar
  lcd.print("Modo:" ); // Que imprimir
  lcd.setCursor(6,0);
  lcd.print(MarcaModo);

  lcd.setCursor(0,1);
  lcd.print("Bateria" );

  if (porcentCargador < 100  && porcentCargador > 0) // Asegurar que la pantalla no imprime valores causados por el ruido
  {
   lcd.setCursor(8,1);
   lcd.print(porcentCargador); // Se el valor dado por Caragdor() no es causado por el ruido, imprimir el valor
  }
  else
  {
   lcd.setCursor(9,1);
   lcd.print("Espera"); // Imprimir esperar si el valor dado por Caragdor() es causado por el ruido
  }

  // MOSTRAR DATOS QUE DEPENDEN DEL MODO----------------------------------------------
  if (MarcaModo == 1)
  {
    lcd.setCursor(9,0);
    lcd.print("Temp:" );
    lcd.setCursor(14,0);
    lcd.print(Input);
  }
}
```

```
void ControlRemoto(int anguloDeseado)
{
  delay(1000); // Asegurar que el señal mandado no tiene ruido de otras partes del Arduino

  long int pulso_high =  map(anguloDeseado, 0, 180, 800, 1000); // Hacer map del angulo para obtener la anchura de pulso

  MarcaControlRemoto = 1;

  long tref_pulso = millis(); // Mandar el pulso HIGH por el tiempo predeterminado en el paso anterior

while(millis()-tref_pulso <= pulso_high)
  {
    if (MarcaCambiarAngAbajo == 1)
    {
      digitalWrite(pinControlServoAb, HIGH);
    }
    else
     {
       digitalWrite(pinControlServoEn, HIGH);

     }
  }

  long int pulso_low = 1000-pulso_high;

  long tref_pulso2 = millis(); // Mandar el pulso LOW por el tiempo predeterminado en el paso anterior

 while(millis()-tref_pulso2 <= pulso_low)
  {
     if (MarcaCambiarAngAbajo == 1)
    {
```

```
      digitalWrite(pinControlServoAb, LOW);
    }
    else
     {
       digitalWrite(pinControlServoEn, LOW);
     }
  }
  tref_pulso2 = millis();

  MarcaControlRemoto=0; // Hacer reset del MarcadorControlRemoto para esperar mas comandos del Pi
  MarcaCambiarAngAbajo = 0;
  MarcaCambiarAngEncima = 0;

}
```

```
dht11 DHT;

float Temperatura()
{
 // USAR EL SENSOR DHT11 PARA OBTENER TEMPERATURA----------------------------------------------
 DHT.read(tempPin); // Inicializar leer temperatura
 temperatura = DHT.temperature; // Leer la temperatura
 if (temperatura > 50) // Si el ruido causa leidas enormes, se usa delay para esperar hasta que estabeliza el sensor
  {
    delay(50);
  }
  else
  {
    Input = .5*temperatura + 0.5*Input; // Un filtro para aumentar suavidad de leida de la temperatura.
  }                                     // Esto también reduce conmutaciones del actuador
  return Input;
}
```

// IMPORTANTE: El ventilador funciona al revez con un pequeño márgen de funcionamiento: pwm de 190 es la velocidad mínima mientras que 0 es la máxima

---

```
void Ventilador(int Speed)
{
 // USAR PWM PARA CONTROLAR EL VENTILADOR-------------------------------------------------------
 if (Speed > 190)
 {
  digitalWrite(pinVentilador, HIGH);
 }
 else
  analogWrite(pinVentilador, Speed); //(pin, speed) // 190 = velocidad mínimna
}
```

ARDUINO SERVO

```
// LIBRERIAS INCLUIDAS Y USADAS----------------
#include <Servo.h> // Incluir Servo library

// VARIABLES------------------------------- LIMITES DE SERVOS
Servo top; // Servo arriba
int servot = 5;  // Posición de servo encima por defecto
const int servotLimitHigh = 80;
const int servotLimitLow = 5;
```

```
Servo bottom;   // Dervo abajo
int servob = 90;  // Posición de servo encima por defecto
const int servobLimitHigh = 180;
const int servobLimitLow = 0;

// VARIABLES----------------------------------DE ELEGIR MODO
int MarcaModo = 0; // 0=Modo Ahorro; 1=Modo Normal

// VARIABLES--------------------------------- SERVO MANUAL
int valorHorizontal;
#define potenciometroTop A5
#define potenciometroBottom A4
int potenciometroValorTop;
int potenciometroValorBottom;

// VARIABLES--------------------------------- SERVO AUTOMATICO
#define photo0 A0 // Se usa const porque esos valores nunca van a cambiar
#define photo1 A1
#define photo2 A2
#define photo3 A3

// VARIABLES----------------------------------PI MODO REMOTO
#define pinControlServoAb 7 // Pin que recibe pulsos del pi que decide el angulo a que mover
#define pinControlServoEn 12 // Pin que recibe pulsos del pi que decide el angulo a que mover

const int pinControlModoArduino = 2; // Pin que recibe pulsos que decide Marcador Modo

int MarcaPi = 1; //1 = Pi control ON, 0 = Pi control OFF

int finalAnguloPi = 0; // Angulo de servo abajo por defecto cuando está en modo Pi
int *finAngPointer; // El pointer que refiere al angulo mandado de Arduino PID (que originalmente viene del Pi)

int finalAnguloPi2 = 10; // Angulo de servo arriba por defecto cuando está en modo Pi
int *finAngPointer2; // El pointer que refiere al angulo mandado de Arduino PID (que originalmente viene del Pi)


void setup()
{
  // SETUP INTERRUPCION de ELEGIR MODO-----------------------------------------------------
  attachInterrupt(digitalPinToInterrupt(3), ElegirModoPi, RISING); // Añadir interrupción a pin 3 que decide entre ServoManual y
                                                   // ServoPi dentro de ModoAhorro.
                                                   // El botón físico tiene prioridad.

  // SETUP IMPRIMIR------------------------------------------------------------------------
  Serial.begin(9600);

  // SETUP SERVOS--------------------------------------------------------------------------
  top.attach(10);
  bottom.attach(9);

  // SETUP MODOS---------------------------------------------------------------------------
  pinMode(pinControlModoArduino, INPUT); // Este pin recibe el pulso que cambia el Modo Ahorro a Modo Normal y al revez
  pinMode(pinControlServoAb, INPUT); // El pulso que controla el Servo Abajo es un input
  pinMode(pinControlServoEn, INPUT); // El pulso que controla es Servo Encima es un input

  pinMode(3, INPUT); // Este pin es una interrpución que cambia MarcaPi.
                 // MarcaPi = 0, ServoManual; MarcaPi = 1, Arduino Servo espera comandos de Pi

  finAngPointer = &finalAnguloPi; // Setup del pointer del angulo abajo mandado de Arduino PID
  finAngPointer2 = &finalAnguloPi2; // Setup del pointer del angulo encima mandado de Arduino PID
}


void loop()
```

```
{
  if (digitalRead(pinControlModoArduino) == HIGH) // El modo del Arduino Servos sólo puede estar cambiado por el pulso del Arduino PID
  {
    MarcaModo = 1; // El modo está cambiado con una marca
  }
  else
  {
    MarcaModo = 0;
  }
  // ELEGIR MODO-------------------------------------------------------------------------
  if (MarcaModo == 1)
  {
    ModoNormal();
  }
  else
  {
    ModoAhorro();
  }
}
```

```
long int last_interrupt_time = millis();

void ElegirModoPi()
{
  if (millis() - last_interrupt_time > 0) // Si la interrupción es más rapida que 100ms, se considera como el bounce y lo se ignora
  {
    Serial.print("Elegir modo pi");
    if (MarcaPi == 1) // Se usa marcadores para indicar el modo.
    {
      MarcaPi = 0;
      Serial.println(MarcaPi);
    }
    else
    {
      MarcaPi = 1;
      Serial.println(MarcaPi);
    }
    last_interrupt_time = millis();
  }
}
```

```
void ModoAhorro()
{
  // Llamar a ServoAhorro()-----------------------------------------------------------
  if (MarcaPi == 0)
  {
    ServoManual();
  }
  else
  {
    // Reciben pulsos de Servo PID y cambian los ángulos correspondientes
    ServoPiAbajo();
    ServoPiEncima();
  }
}
```

```
void ModoNormal()
{
  // Llamar a ServoAutomático()-------------------------------------------------------------
  ServoAutomatico();
}
```

```
void ServoAutomatico()
{
// OBTENER LECTURAS Y ESTABLECER LAS DIFERENCIAS ENTRE ELLOS QUE DECIDIRÁ LA DIRECCIÓN DEL
MOVIMIENTO----------

  // Leer los pins para obtener los datos de cada fotoresistor
  int Intensidad0 = analogRead(photo0);
  int Intensidad1 = analogRead(photo1);
  int Intensidad2 = analogRead(photo2);
  int Intensidad3 = analogRead(photo3);

  int tol = 20; //Determinar la diferencia que va a ser un punto después de que muevan los servos

  // Comparar los valroes en los lados opuestos (encima con abajo, derecho con izquierdo)
  int Dif01 = Intensidad0 - Intensidad1;
  int Dif23 = Intensidad2 - Intensidad3;

  // MOVER MOTOR ARRIBA (vertical)  MIENTRAS QUE HAY DIFERENCIA ENTRE ARRIBA Y ABAJO--------------------------------
  if (-1*tol >  Dif23 || Dif23 > tol) // Asegurar que la diferrencia recibida entre arriba y abajo es significativa
  {
    if (Intensidad2 > Intensidad3) // Si el lado arriba recibe más luz que el lado abajo
    {
      servot = ++servot; // Mover el motor un unido hasta en la dirección de arriba solo hasta llegar a su límite
      if (servot > servotLimitHigh)
      {
        servot = servotLimitHigh; //No hacer nada, porque ya está en el limite
      }
    }
    else if (Intensidad2 < Intensidad3) // Si el lado arriba recibe menos luz que el lado abajo
    {
      servot= --servot; // Mover el motor un unido hasta en la dirección de abajo solo hasta llegar a su límite
      if (servot < servotLimitLow)
      {
        servot = servotLimitLow; // No hacer nada, porque ya está en el limite
      }
    }
    top.write(servot);
  }

  // MOVER MOTOR ABAJO (horizontal)  MIENTRAS QUE HAY DIFERENCIA ENTRE DER. y IZQ.---------------------------------
  if (-1*tol > Dif01 || Dif01 > tol) // Asegurar que la diferrencia recibida entre derecho y recto es significativa
  {
    if (Intensidad0 > Intensidad1) // Si lado izquierdo recibe más luz que el lado derecho
    {
      servob = ++servob; // Mover el motor un unido a la izquierda si todavía no llegó el servo a su limite
      if (servob < servobLimitLow)
      {
        servob = servobLimitLow; // No hacer nada, porque ya está en la limite
      }
    }
    else if (Intensidad0 < Intensidad1)
    {
      servob = --servob; // Mover el motor un unido a la derecha si todavía no llegó el servo a su limite
      if (servob > servobLimitHigh) // Si lado izquierdo recibe menos luz que el lado derecho
```

```
    {
      servob = servobLimitHigh; // No hacer nada, porque ya está en la limite
    }
   }
   bottom.write(servob);
 }
 //IMPRIMIR LOS VALORES DE FOTORESISTORES PARA ASEGURAR FUNCIONAMIENTO----------------------------------------
 Serial.print("Photo 0:");
 Serial.print(Intensidad0);
 Serial.print(" ");
 Serial.print("Photo 1:");
 Serial.print(Intensidad1);
 Serial.print(" ");
 Serial.print("Photo 2:");
 Serial.print(Intensidad2);
 Serial.print(" ");
 Serial.print("Photo 3:");
 Serial.println(Intensidad3);
 //Serial.print("\n");
}
```

```
void ServoManual()
{
 // USAR ENTREADA ANALÓGICA PARA MOVER EL SERVO-------------------------------------------------------------
 potenciometroValorTop = analogRead(potenciometroTop); // Leer los datos del potenciometro
 servot = map(potenciometroValorTop, 0, 1023, 5, 80); // Asgignar los datos leido entre los
                                                      // limites del servo que está encima
 top.write(servot);  // Mover el servo hasta llegar al valor indicado en el paso anterior

 potenciometroValorBottom = analogRead(potenciometroBottom); // Leer los datos del potenciometro
 servob = map(potenciometroValorBottom, 0, 1023, 0, 180); // Asgignar los datos leido entre los
                                                          // limites servo que está abajo.
 bottom.write(servob); // Mover el servo hasta llegar al valor indicado en el paso anterior

 Serial.println(potenciometroValorTop);
 Serial.println(potenciometroValorBottom);
}
```

```
float anchoPulso;
float anguloPiAb;
int anguloPrevioAb; // Siempre empezar con el angulo 90 de servo abajo

void ServoPiAbajo()
{
 long int tref_pulso = millis();
 while(digitalRead(pinControlServoAb) == HIGH) // Recibir el puslo del Arduino PID
 {
  long int tiempo_corriente = millis();
  if(tiempo_corriente-tref_pulso < 1000 && tiempo_corriente-tref_pulso > 800) // Ignorar ruido que puede venir de otro Arduino
  {
   anchoPulso = tiempo_corriente-tref_pulso;
  }
  anguloPiAb = map(anchoPulso, 800, 1000, 0, 180); // Hacer map para obtener el angulo
  *finAngPointer = anguloPiAb;
 }

  anguloPrevioAb = bottom.read(); // Leer el angulo previo del servo para monitorizar/debug

 //Serial.print("anguloPrevio:");
 //Serial.println(anguloPrevioAb);
```

```
  //Serial.print("funalAnguloPi:");
  //Serial.println(finalAnguloPi);


   if (anguloPrevioAb > finalAnguloPi) // Comparar el valor del angulo que está asociado
                                       // con el pointer con el valor previo del servo abajo
    {
      servob = --servob; // Mover el motor un unido hasta en la dirección de arriba
                         // solo hasta llegar a su límite
      if (servob > servotLimitHigh)
      {
        servob = servotLimitHigh; // No hacer nada, porque ya está en el limite
      }

      //Serial.println(anguloPrevioAb); // Mover 1 unidad
      bottom.write(servob);
      anguloPrevioAb = bottom.read();
    }
  else if (anguloPrevioAb < finalAnguloPi)  // Comparar el valor del angulo que está asociado
                                            // con el pointer con el valor previo del servo abajo
    {
      servob = ++servob; // Mover el motor un unido hasta en la dirección de abajo
                         // solo hasta llegar a su límite
      if (servob < servotLimitLow)
      {
        servob = servotLimitLow; // No hacer nada, porque ya está en el limite
      }
      // Serial.println(anguloPrevioAb);
      bottom.write(servob);
      anguloPrevioAb = bottom.read();
    }
}
```

```
int anguloPrevioEn; // Siempre empezar con el angulo 90 de servo abajo
float anguloPiEn;

void ServoPiEncima()
{
 long int tref_pulso2 = millis();
  while(digitalRead(pinControlServoEn) == HIGH) // Recibir el puslo del Arduino PID
  {
    long int tiempo_corriente = millis();
    if(tiempo_corriente-tref_pulso2 < 1000 && tiempo_corriente-tref_pulso2 > 800) // Ignorar ruido que puede venir de otro
                                                                                  // Arduino
    {
     anchoPulso = tiempo_corriente-tref_pulso2;
    }
    anguloPiEn = map(anchoPulso, 800, 1000, 0, 180); // Hacer map para obtener el angulo
    *finAngPointer2 = anguloPiEn;
  }

  anguloPrevioEn = top.read(); // Leer el angulo previo del servo para monitorizar/debug
  Serial.print("anguloPrevio:");
  Serial.println(anguloPrevioEn);

  Serial.print("funalAnguloPi:");
  Serial.println(finalAnguloPi);

   if (anguloPrevioEn > finalAnguloPi2) // Comparar el valor del angulo que está asociado con el pointer con el valor previo del
                                        // servo abajo
    {
```

```
    servot = --servot; // Mover el motor un unido hasta en la dirección de arriba solo hasta llegar a su límite
    if (servot > servotLimitHigh)
    {
      servot = servotLimitHigh; // No hacer nada, porque ya está en el limite
    }

    Serial.println(anguloPrevioEn); // Mover 1 unidad
    top.write(servot);
    anguloPrevioEn = top.read();
  }
 else if (anguloPrevioEn < finalAnguloPi2)  // Comparar el valor del angulo que está asociado con el pointer con el valor previo
                                            // del servo abajo
  {
    servot = ++servot; // Mover el motor un unido hasta en la dirección de abajo solo hasta llegar a su límite
    if (servot < servotLimitLow)
    {
      servot = servotLimitLow; // No hacer nada, porque ya está en el limite
    }
    Serial.println(anguloPrevioEn);
    top.write(servot);
    anguloPrevioEn = top.read();
  }
}
```

4.3 REFERENCIAS

FUENTE DE ALIMENTACIÓN

USB información general:

http://www.frozencpu.com/products/17518/ele-1137/USB_30_Type-A_9-Pin_Male_External_Connector.html

https://www.open-electronics.org/the-power-of-Arduino-this-unknown/

Arduino cable info:

http://forum.Arduino.cc/index.php?topic=132448.0

SERVOS

Servo funcionamiento:

https://en.wikipedia.org/wiki/Servo_control

https://www.pololu.com/blog/20/advanced-hobby-servo-control-pulse-generation-using-hardware-pwm

http://www.the-diy-life.com/Arduino-solar-tracker/

http://forum.Arduino.cc/index.php?topic=45000.0

Potenciometro:

https://programarfacil.com/blog/el-potenciometro-y-Arduino/

http://panamahitek.com/que-es-y-como-funciona-un-potenciometro/

PID

PID General:

http://digital.ni.com/public.nsf/allkb/4A0881274E4CC2A086256C3600577548

https://Arduino.stackexchange.com/questions/9731/control-speed-of-dc-fan-using-Arduino-pid-library

https://rydepier.wordpress.com/2016/05/30/l9110-fan-motor-keyes-board/

Debounce:

http://forum.arduino.cc/index.php?topic=45000.0

Otras sistemas de enfriamiento:

http://www.sciencedirect.com/science/article/pii/S2090447913000403

NTC sensor:

https://en.tdk.eu/download/531110/a3be527165c9dd17abca4970f507014f/pdf-applicationnotes.pdf

https://es.wikipedia.org/wiki/Termistor

CARGADOR

Batería:

http://www.scarpaz.com/Attic/Documents/NiMH_technical.pdf

https://en.wikipedia.org/wiki/Electrolysis_of_water

https://en.wikipedia.org/wiki/Nickel%E2%80%93metal_hydride_battery

http://www2.emersonprocess.com/siteadmincenter/PM%20Asset%20Optimization%20Documents/ProductBrochuresAndFlyers/375_br_nickelmetalhydride.pdf

https://www.nickelinstitute.org/NickelUseInSociety/AboutNickel/WhereWhyNickelIsUsed.aspx

http://www.powerstream.com/NiMH.htm

https://rc.runryder.com/helicopter/t330664p1/

http://www.candlepowerforums.com/vb/showthread.php?391127-Can-3x-AAA-NiMH-cells-deliver-3-Ampere-for-an-XP-L

http://data.energizer.com/pdfs/nickelmetalhydride_appman.pdf

http://www.ebah.com.br/content/ABAAABVHAAF/eletronica

Peligro de sobrecargar:

https://www.rcgroups.com/forums/showthread.php?373602-Fire-danger-from-Ni-Cd-s-and-NiMH-s-like-there-is-from-Li-Poly

Panel solar general info:

https://curiosoando.com/como-funciona-un-panel-solar

http://www.robotshop.com/blog/en/how-do-i-choose-a-battery-8-3585

http://www.mrsolar.com/content/faqs/when-do-I-need-a-charge-controller-and-why.html

Diodos:

http://www.monografias.com/trabajos-pdf2/diodo-schottky-barrera/diodo-schottky-barrera.shtml

https://estudiaparalavida.jimdo.com/diodos-de-barrera-schotky/

https://es.wikipedia.org/wiki/Diodo_Schottky

ISR:

https://barrgroup.com/Embedded-Systems/How-To/C-Volatile-Keyword

http://www.engblaze.com/microcontroller-tutorial-avr-and-Arduino-timer-interrupts/

https://www.youtube.com/watch?v=m5_pFID-f-M

https://www.youtube.com/watch?v=U6FLgeLV3j8

Transistor:

https://learn.sparkfun.com/tutorials/transistors/applications-i-switches

https://es.wikipedia.org/wiki/Transistor

Pi:

http://www.instructables.com/id/Raspberry-Pi-Arduino-Serial-Communication/

ESPECIFICACIONES:

Sensor Temperatura:

http://www.uugear.com/portfolio/dht11-humidity-temperature-sensor-module/

Servo:

Pdf bajado

Funcionamiento: https://learn.sparkfun.com/tutorials/hobby-servo-tutorial

Dimensiones: https://www.addicore.com/Addicore-SG90-Mini-Servo-p/113.htm

Ventilador:

http://www.miniinthebox.com/es/modulo-de-ventilador-keyes-l9110-para-Arduino_p2373651.html

Pantalla:

https://www.openhacks.com/uploadsproductos/eone-1602a1.pdf

Fotoresistencia:

http://akizukidenshi.com/download/ds/senba/GL55%20Series%20Photoresistor.pdf

Funcionamiento: https://electronica-electronics.com/info/LDR-fotoresistencia.html

http://www.reuk.co.uk/wordpress/electric-circuit/light-dependent-resistor/

Potenciómetro:

https://www.sparkfun.com/datasheets/Components/General/R12-0-.pdf