adafruit learning system

Sous-vide controller powered by Arduino - The SousViduino!

Created by Bill Earl

Last updated on 2015-01-03 06:45:16 AM EST

Guide Contents

What is Sous Vide?

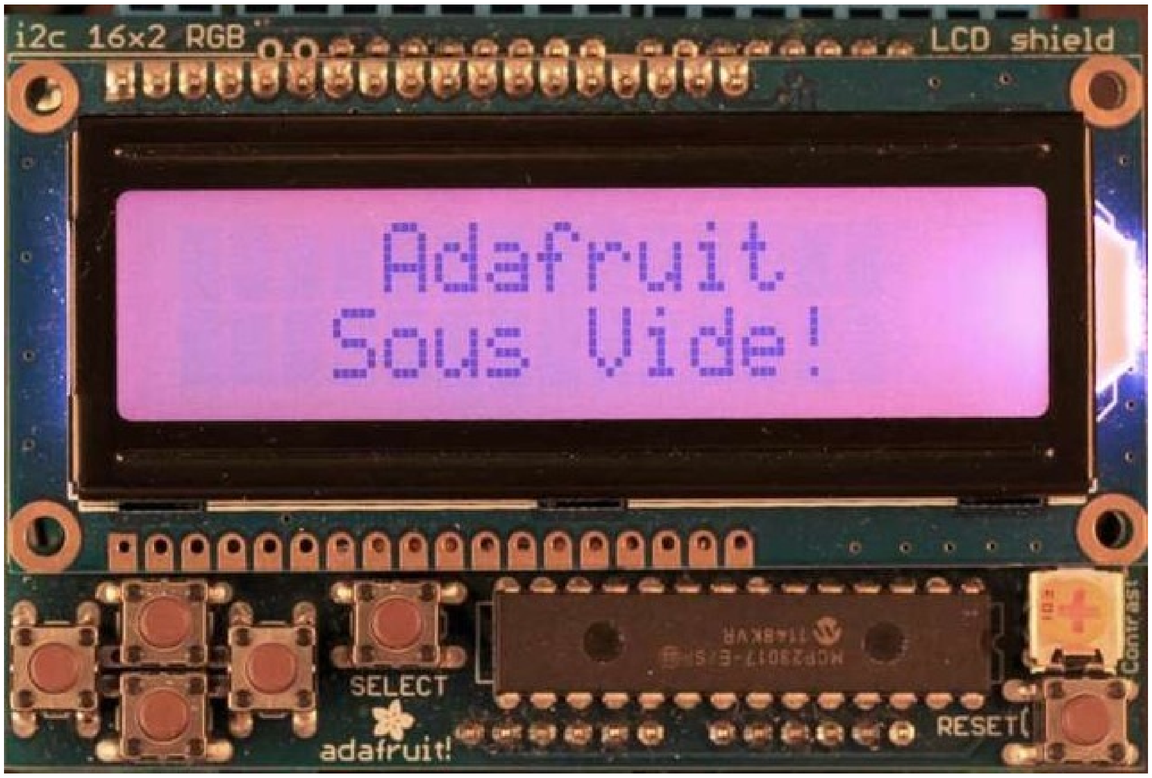
> *"...far from being some passing high-tech fad, sous vide is a lasting contribution to fine cooking, a technique that makes it possible to cook foods more consistently and delicately than ever before."*
> *(from "Under Pressure" by Thomas Keller)*

Want to make delicious, perfectly-cooked food using a ***robot***? Who doesn't!? This project will show you how to build your own "Sous viduino", an automated cooking pot that makes perfect eggs, succulent steaks and tender fish without the whole "slaving over a stove." All this is possible due to a recent advancement in cooking technology - instead of using a pan or pot, a "sous vide" (pronounced *suu veed*) machine heats the food in sort of a cross between a jacuuzi and a crock pot.

Sous vide is rapidly becoming an important cooking technique in many of the very best restaurants in the world. Sous vide combines principles of molecular gastronomy with industrial temperature controls to precisely manage the chemical reactions of cooking.

We love good food as much as we love science and technology, so of course we had to build our own sous vide controller. This project turns an inexpensive rice cooker into a precision cooking instrument capable of maintaining cooking temperatures within +/-0.1 C.

By precisely controlling the temperature, you can ensure that foods are cooked exactly to the desired level of doneness and no-more. This makes sous vide the preferred cooking method for everything from the 'perfect' soft-boiled egg to the steak that is medium-rare from edge-to-edge.



Materials

The Controller

You don't need a powerful microcomputer to drive this setup. We selected the Arduino for this project because of the excellent (and well documented) PID and autotune libraries available for it. The Adafruit RGB/LCD shield with integral pushbuttons was also perfect for hosting the user interface.

To build this controller, you will need:

- Arduino UNO (http://adafru.it/50)
- Proto Shield (http://adafru.it/51) or Wing Shield (http://adafru.it/196)
- RGB LCD Shield (Positive (http://adafru.it/716) or Negative (http://adafru.it/714)display)
- Power Switch Tail (http://adafru.it/268)
- High Temperature DS18B20 Temperature Sensor (http://adafru.it/642) you can also use the standard Waterproof DS18B20 type (http://adafru.it/381) if you aren't planning on heating it past 90C, which is very rare for Sous Vide
- Food Grade Heat-Shrink Tubing (http://adafru.it/1020)
- Servo Extension Cable (http://adafru.it/972)
- In-Line JST power wire connector: Male (http://adafru.it/319)and Female (http://adafru.it/318)

We have all the above as part of a special discounted pack http://www.adafruit.com/products/1401 (http://adafru.it/1401)

You will also need some basic tools and soldering supplies:

- WIre cutters
- Wire strippers
- Soldering Iron
- Solder
- Shrink Wrap Tubing
- Small Phillips Screwdriver

Cooker Selection

Type

Rice Cookers and Slow Cooker appliances have the basic elements we are looking for:

- A cooking vessel that holds water
- An electric heating element

Capacity

You need a vessel substantially larger than the food you intend to cook in it. It needs to have a large enough volume of water with plenty of room for circulation to maintain an even temperature. Look for a rice cooker with a capacity of **at least** 10 cups uncooked (20 cups cooked) or a slow cooker with **at least** a 4 quart (4 liter) capacity.

Controls

Since we are building our own controller, you want the simplest manual control cooker you can find. One with a simple switch or control knob is best. We will just turn it on at the highest setting and plug it into our controller. (Here at Adafruit we picked up a "Black and Decker RC880" - a rice cooker with 24 cup capacity and is a great size)

Cookers with digital controls are not suitable - unless you want to rip them apart to bypass the electronics. The controller works by pulsing the power to the cooker. If you do that with a digitally controlled cooker, it will just turn off!

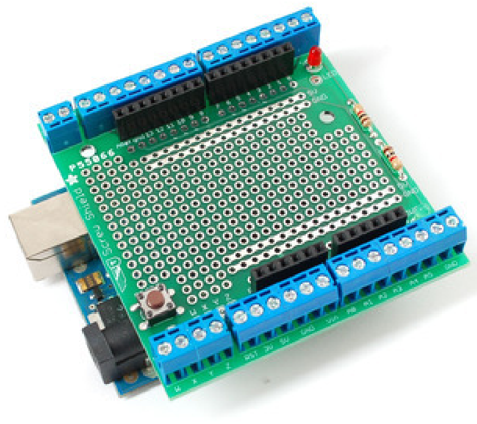
Build the Controller

Prepare the Sensor.

Most sous vide cooking is done in sealed plastic bags. One exception to this is eggs, which are cooked in their shells. Other recipes (such as butter-poached lobster tails) call for the food to be immersed directly in the cooking liquid. To make the sensor safe for contact with food, cut a length of food-safe heat shrink tubing and shrink it around the sensor as below.

Build the RGB LCD Shield

Follow the instruction in this tutorial (http://adafru.it/ceP) to build and test the shield.



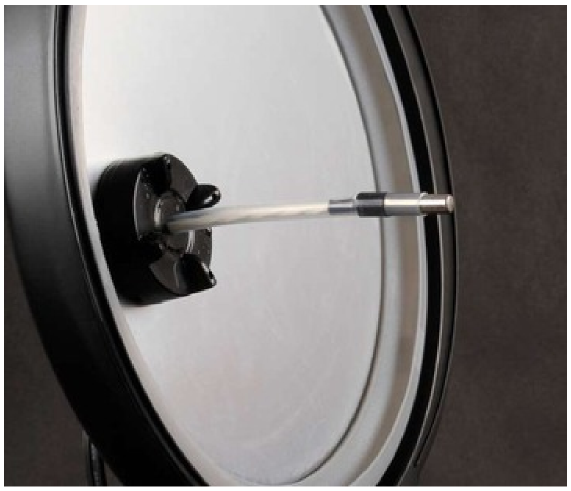
Build the Wing Shield

Follow the instructions in this tutorial (http://adafru.it/cnJ) to build the wing shield.

Note that the switch, leds and resistors are not used for the sous vide controller and don't need to be installed. You can save these parts for your next project.

Prepare the sensor

Our waterproof DS18B20 sensors are great for immersing in liquid and measuring temperature, but they are not food safe on their own. If you are only going to be cooking food in a plastic bag, this doesn't matter so much but it will matter if you cook food straight in the water like eggs. We strongly suggest going the extra step and using some food safe heat-shrink to cover everything but the stainless-steel tip of the sensor. Use a hot-air gun to shrink the sleeve water-tight over the sensor.

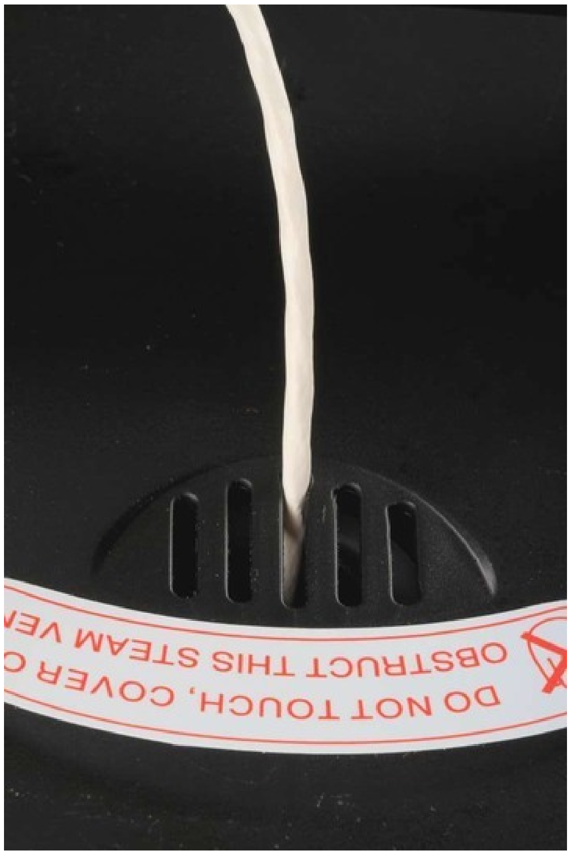


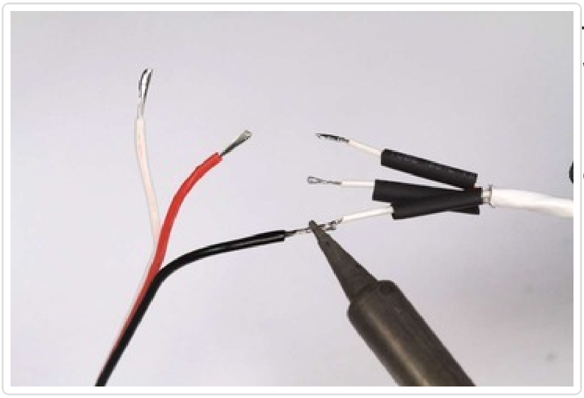
Install the Sensor

Drill or enlarge the steam vent opening so that the sensor wire can be routed through the lid. Position the sensor so that it extends about midway down into the cooking vessel when the lid is closed.

If it is a loose fit. Put a cable-tie around the sensor wire where it exits the lid to prevent it from sliding deeper into the cooker.

Add additional cable ties every 4-6" to fasten the sensor wire to the power cord.





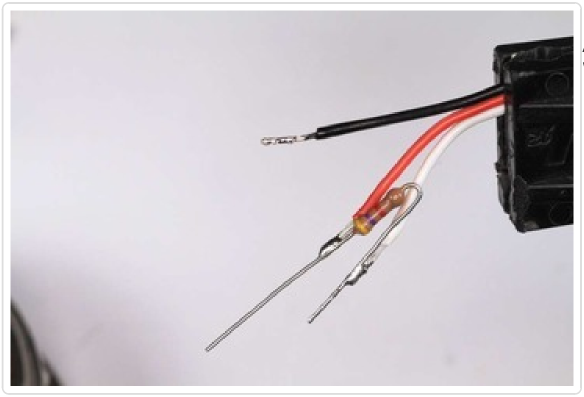
Terminate the Sensor Wires

We'll use a servo extension cable as a way of making a detachable sensor. Its not a requirement but it makes it easier to set up and clean up.

- Strip and prepare some lengths of heat-shrink.
- Cut a servo extension cable in half.
- Solder the male end to the sensor wires.
- Heat-shrink to insulate.

The sensor wire color coding (on the right) is a little strange, so we'll connect them up to use a more standard color coding for the servo connector (right).

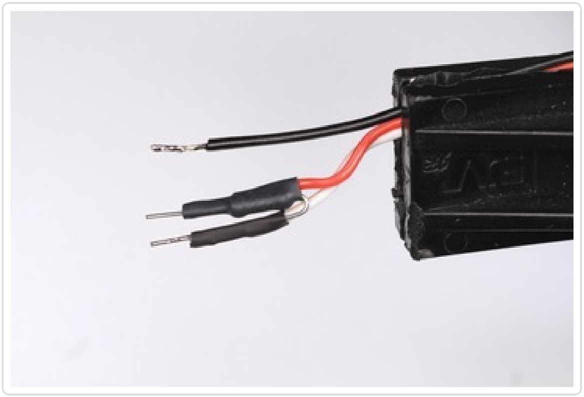
- **Black** <- White (ground)
- **Red** <- Orange Stripe (+5v)
- **White** <- Blue Stripe (Signal)

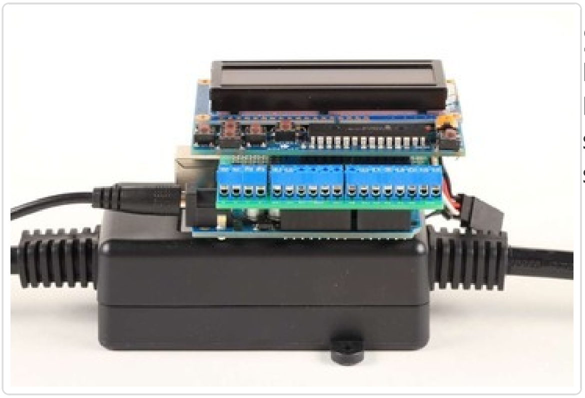


Add the Resistor

With the female half of the servo extension:

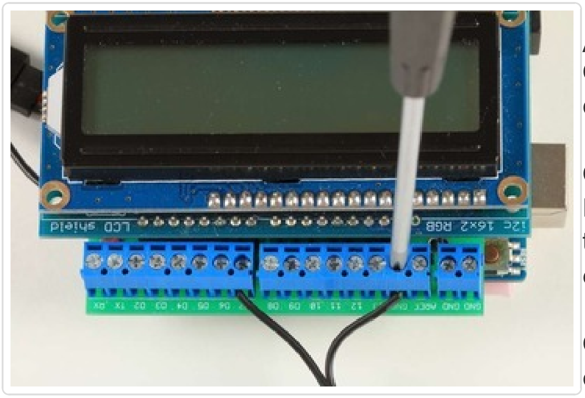
- Strip and tin the ends
- Solder the 4.7K ohm resistor that came with your sensor between the 5v (Red) and signal (White) wires.
- Heat shrink to insulate and trim.

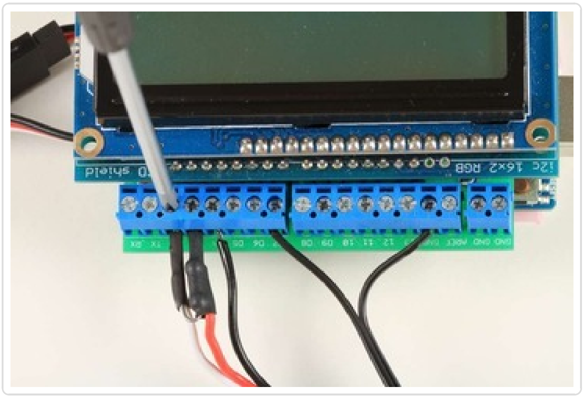


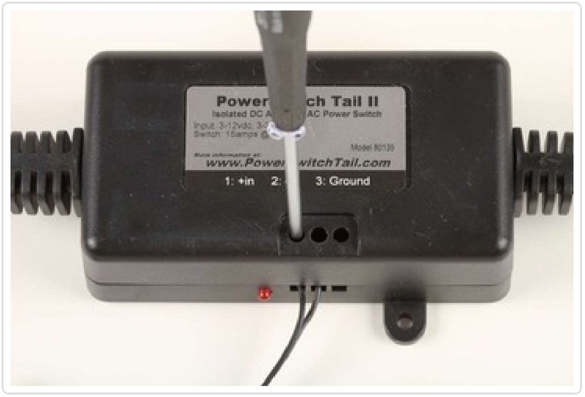


Stack them up!

Plug the Wing Shield into the Uno and the RGB LCD Shield into the Wing Shield. Attach the stack on top of the PowerSwitch Tail and add some cable ties for security.







Attach the cables

Connect male JST power cable to pins 7 and ground of the wing shield.

Connect the female JST cable to the PowerSwitch Tail terminal screws. Be sure that the polarity is the same as it is on the other end.

Connect the male half of the servo extension cable:

- **White** -> Pin 2 (Signal)
- **Red** -> Pin 3 (5v)
- **Black** -> Pin 4 (Ground)

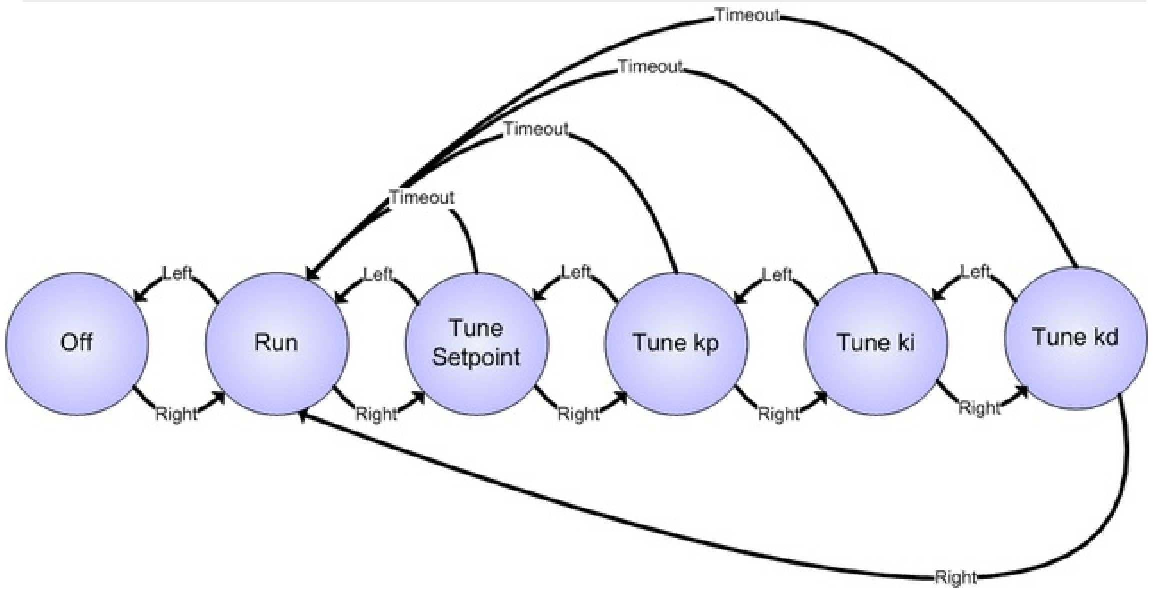
Put it all together:

- Connect the male and female ends of the servo extension cable.
- Connect the male and female JST cables together.
- Plug the cooker into the PowerSwitch Tail
- Plug the PowerSwitch Tail into a wall outlet.

Now you are ready to load some software!

Adafruit
Sous Vide!

Control Software



The following pages walk you through various aspects of the controller code, explaining how each one works. The final page of this section contains the complete controller sketch, ready for upload to your sous vide controller.

PID

Before we can get cooking we have to solve a simple-sounding problem: how to keep the temperature of the water in the cooker at a set temperature for a long time. Its really easy to get water to boiling (just heat it until it starts to boil) or freezing (cool it till it solidifies) but keeping a water bath stable is a little tougher. The water cools off as it sits, and as it heats up the food, but how fast it cools depends on the amount of water, the temperature in the room, the temperature of the food and how much there is. Its hard to do by 'hand' we will automate it using **PID Feedback Control**

What's a PID?

We need to have our microcontroller control the heater to keep the temperature stable. It can heat the water by turning on the rice cooker (**control**), and it can measure the temperature with our waterproof sensor (**feedback**). The stuff in the middle is the the algorithm that ties these together. The PID algorithm is a type of feedback control. In this application, the temperature **measurement** is compared with the **setpoint** and the difference between them is called the **error**. The **error** is used to calculate an adjustment to the **output** which controls the heating element.

The PID name comes from the three terms in the equation used to calculate the output:

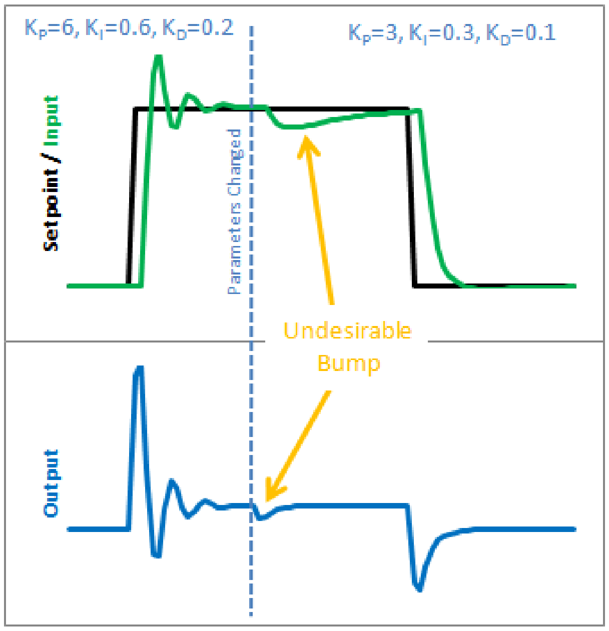
- **P** - The **Proportional** term looks at the present state of the process. Its value is proportional to the current error.
- **I** - The **Integral** term looks at the history of the process. Its value is the integral of past errors.
- **D** - The **Derivative** tries to predict the future of the process. Its value is the derivative or rate of change in the error..

These three terms are assigned weights known as **tuning parameters: Kp, Ki** and **Kd.** The three terms are summed to produce the control output.

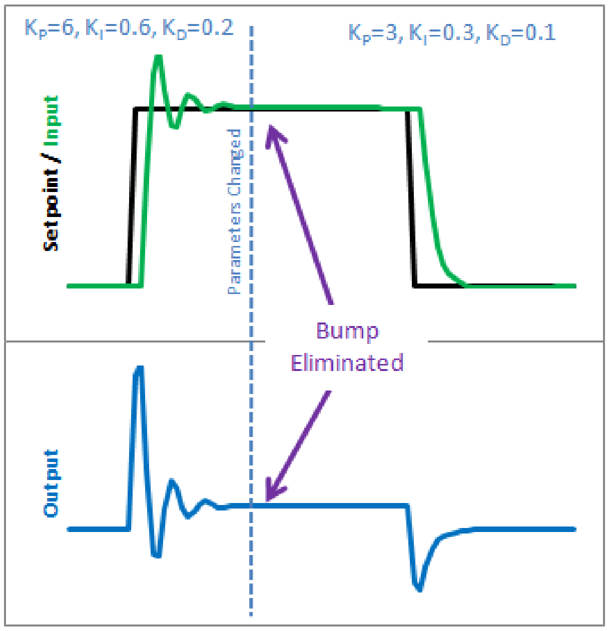
$$\text{Output} = K_P e(t) + K_I \int e(t)\,dt + K_D \frac{d}{dt} e(t)$$

$$\text{Where : } e = \text{Setpoint - Input}$$

The basic PID equation is not that difficult to implement and there are many PID implementations out there. But there are a lot of 'gotchas' in making a PID perform well in a real-world application.



These gotchas have been expertly addressed by Brett Beauregard in his Arduino PID Library (http://adafru.it/ceR). And clearly documented in his blog post: Improving the Beginners PID (http://adafru.it/ceS).

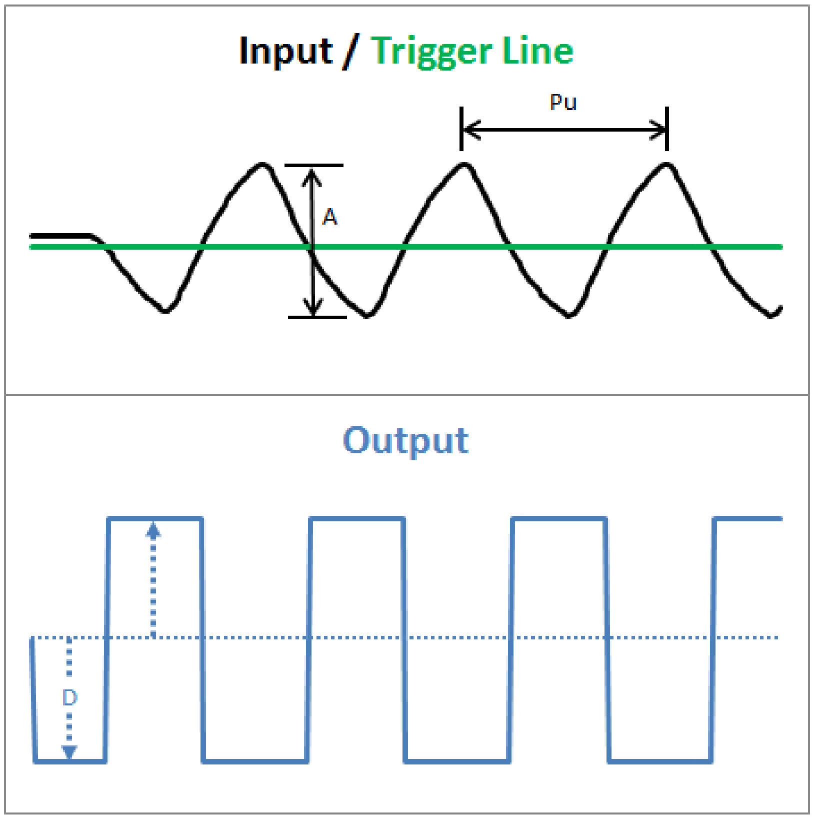


Thanks to Brett Beauregard for permission to use these images.

Autotune

You might have heard of "Auto Tuning" as a way to filter a singing voice to hit perfect pitches. Auto-tuning a PID controller is not quite the same thing - instead of improving your mediocre singing voice, it can help you to set the initial tuning parameters for your controller. Every 'system' has different parameters inherent in the physical world. For example, the sous vide controller has to account for how many Watts of power the rice cooker uses, how fast it takes to start up, the specific heat of water, etc.

The autotune function attempts to characterize the performance of your sous vide system by disrupting (essentially 'poking') the controller output to see how it responds. This is sort of like poking your cat to see about how much prodding he will take before you get the response you desire (say, jumping off the table) but not so much that you get a bad response (say, scratching you)



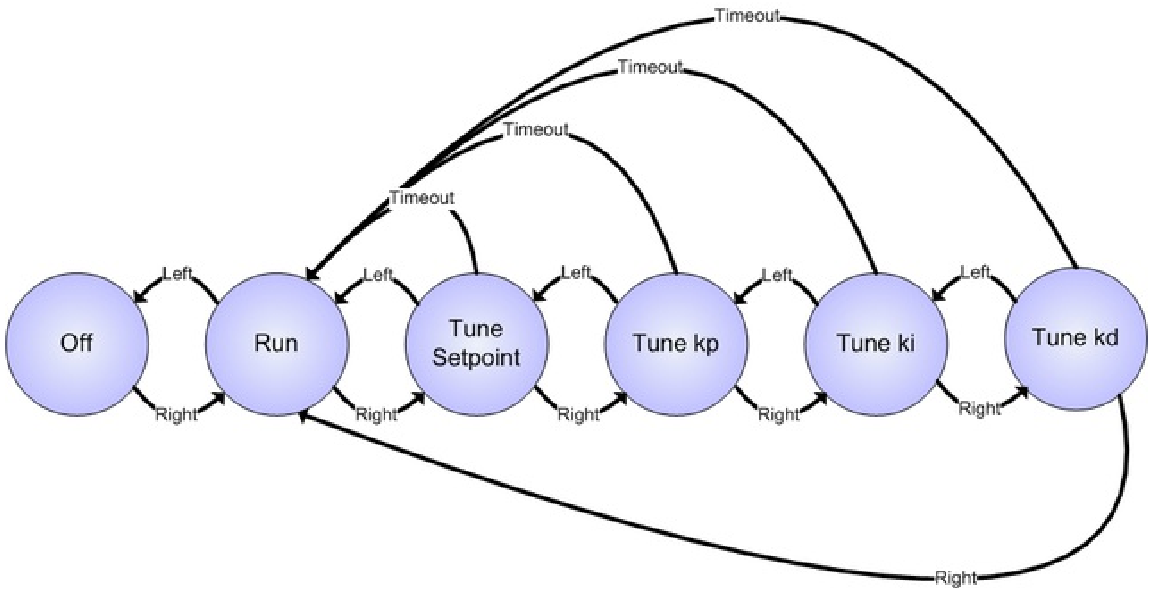
Based on the response to the disruption, the algorithm calculates the Kp, Ki and Kd tuning parameters. Autotune is not perfect and may not find the optimal tuning for your system, but it should get you in the ballpark.

For more detail on how the autotune algorithm works, read Brett's blog post on the topic here: Arduino PID Autotune Library (http://adafru.it/ceT)

User Interface

The sous vide user interface allows you to set cooking temperatures and make adjustments to the PID tuning parameters. It is implemented as a simple state machine. Each state implements a user interface 'screen'. State machines are often used in semi-complex microcontroller projects where you want to do a lot of configuring and activities in the correct order.

The shield buttons are used to navigate between the different screens. After a period of inactivity, a timeout returns the system to the "Run" screen to display the current set-point (the temperature we desire) and current temperature of the bath. The states are as shown in the diagram below:



Each state is implemented as a function that is called by the main loop based on the **opState** variable. To change states, a state function sets **opState** to the new state and returns to the main loop.

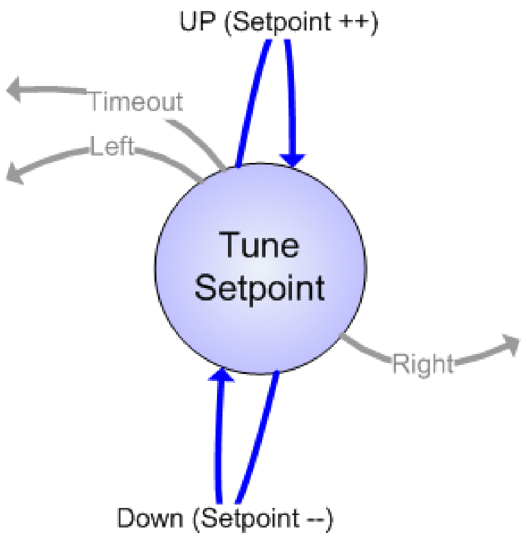
If you're working on a project that has a lot of stuff going on, drawing a state machine can be *really* useful to keep your head straight!

```
// ************************************************
// Main Control Loop
//
// All state changes pass through here
// ************************************************
void loop()
{
   // wait for button release before changing state
   while(ReadButtons() != 0) {}

   lcd.clear();

   Serial.println(opState);
```

```
  switch (opState)
  {
  case OFF:
     Off();
     break;
  case SETP:
     Tune_Sp();
     break;
   case RUN:
     Run();
     break;
  case TUNE_P:
     TuneP();
     break;
  case TUNE_I:
     TuneI();
     break;
  case TUNE_D:
     TuneD();
     break;
  }
}
```

Each state function is responsible for updating the display and monitoring button presses. In addition to navigating between screens, buttons presses are also used to modify the control parameters. For example, the **Tune Setpoint** state uses the **UP** and **DOWN** keys to modify the setpoint as shown in the diagram and in the code below.

The 5th button on the shield is used as a 'shift' key. When pressed simultaneously with the **UP** or **DOWN** keys, it increments or decrements by 10 instead of just 1. The other tuning screens work the same way.



```
// ************************************************
// Setpoint Entry State
// UP/DOWN to change setpoint
// RIGHT for tuning parameters
// LEFT for OFF
// SHIFT for 10x tuning
// ************************************************
void Tune_Sp()
{
   lcd.setBacklight(TEAL);
   lcd.print(F("Set Temperature:"));
   uint8_t buttons = 0;
   while(true)
   {
      buttons = ReadButtons();

      float increment = 0.1;
      if (buttons & BUTTON_SHIFT)
      {
        increment *= 10;
      }
      if (buttons & BUTTON_LEFT)
      {
         opState = RUN;
         return;
      }
      if (buttons & BUTTON_RIGHT)
      {
         opState = TUNE_P;
         return;
      }
      if (buttons & BUTTON_UP)
      {
         Setpoint += increment;
         delay(200);
      }
      if (buttons & BUTTON_DOWN)
      {
         Setpoint -= increment;
         delay(200);
      }

      if ((millis() - lastInput) > 3000)  // return to RUN after 3 seconds idle
      {
         opState = RUN;
         return;
      }
      lcd.setCursor(0,1);
      lcd.print(Setpoint);
      DoControl();
   }
}
```

Persistent Data

So you don't have to hard-code the tuning parameters or enter them every time you use the controller, we save them in the Arduino's EEPROM. This makes the code a little more elegant as you can program the controller once, then re-tune if you ever change cookers but whenever you turn on your controller, it will remember the settings from the last time you used it.

SInce EEPROM can only be written a finite number of times (typically 100,000), we compare the contents before writing and only write if something has changed. This functionality is implemented in the following helper functions:

```
// ************************************************
// Save any parameter changes to EEPROM
// ************************************************
void SaveParameters()
{
   if (Setpoint != EEPROM_readDouble(SpAddress))
   {
      EEPROM_writeDouble(SpAddress, Setpoint);
   }
   if (Kp != EEPROM_readDouble(KpAddress))
   {
      EEPROM_writeDouble(KpAddress, Kp);
   }
   if (Ki != EEPROM_readDouble(KiAddress))
   {
      EEPROM_writeDouble(KiAddress, Ki);
   }
   if (Kd != EEPROM_readDouble(KdAddress))
   {
      EEPROM_writeDouble(KdAddress, Kd);
   }
}

// ************************************************
// Load parameters from EEPROM
// ************************************************
void LoadParameters()
{
  // Load from EEPROM
   Setpoint = EEPROM_readDouble(SpAddress);
   Kp = EEPROM_readDouble(KpAddress);
   Ki = EEPROM_readDouble(KiAddress);
   Kd = EEPROM_readDouble(KdAddress);

   // Use defaults if EEPROM values are invalid
   if (isnan(Setpoint))
   {
     Setpoint = 60;
   }
   if (isnan(Kp))
   {
     Kp = 500;
   }
   if (isnan(Ki))
   {
     Ki = 0.5;
   }
   if (isnan(Kd))
   {
     Kd = 0.1;
   }
}


// ************************************************
// Write floating point values to EEPROM
// ************************************************
void EEPROM_writeDouble(int address, double value)
{
   byte* p = (byte*)(void*)&value;
   for (int i = 0; i < sizeof(value); i++)
   {
      EEPROM.write(address++, *p++);
   }
}

// ************************************************
// Read floating point values from EEPROM
// ************************************************
double EEPROM_readDouble(int address)
{
   double value = 0.0;
   byte* p = (byte*)(void*)&value;
   for (int i = 0; i < sizeof(value); i++)
   {
      *p++ = EEPROM.read(address++);
   }
   return value;
}
```

Time Proportional Output

Since the cooker's heater is controlled by a relay, we can't use a standard PWM output to control it. PWM is a very easy and precise way to control heating but requires a more expensive SSR. There are some PID feedback systems that benefit greatly from PWM control. Fortunately, our system is a big tub of water and water heats up and cools down very slowly. Due to this 'thermal mass' of the system, the response time is relatively slow so we can use a very slow form of PWM known as "Time Proportional Output". In this case, the frequency of the pulses is 0.1 Hz or once very 10 seconds. Its basically like really really slow-scale PWM

We need to control the pulse timing accurately, so we don't want to be affected by any delays that there might be in the main loop. So we use a timer to generate a periodic interrupt. The timer is initialized in setup():

```
 // Run timer2 interrupt every 15 ms
 TCCR2A = 0;
 TCCR2B = 1<<CS22 | 1<<CS21 | 1<<CS20;

 //Timer2 Overflow Interrupt Enable
 TIMSK2 |= 1<<TOIE2;
```

And the interrupt service routine is called once every 15 milliseconds to update the relay output.

```
// ************************************************
// Timer Interrupt Handler
// ************************************************
SIGNAL(TIMER2_OVF_vect)
{
 if (opState == OFF)
 {
  digitalWrite(RelayPin, LOW);  // make sure relay is off
 }
 else
 {
  DriveOutput();
 }
}
```

The DriveOutput() function implements the time proportional output.

```
// ************************************************
// Called by ISR every 15ms to drive the output
// ************************************************
void DriveOutput()
{
 long now = millis();
 // Set the output
 // "on time" is proportional to the PID output
```

```
  if(now - windowStartTime>WindowSize)
  { //time to shift the Relay Window
    windowStartTime += WindowSize;
  }
  if((onTime > 100) && (onTime > (now - windowStartTime)))
  {
    digitalWrite(RelayPin,HIGH);
  }
  else
  {
    digitalWrite(RelayPin,LOW);
  }
}
```

Putting it all together!

Here is the complete sketch for the Adafruit Sous Vide Controller

You can also get the latest code (which may have updates or improvements) from Github at https://github.com/adafruit/Sous_Viduino (http://adafru.it/ceU)

```
//-------------------------------------------------------------------
//
// Sous Vide Controller
// Bill Earl - for Adafruit Industries
//
// Based on the Arduino PID and PID AutoTune Libraries
// by Brett Beauregard
//------------------------------------------------------------------

// PID Library
#include <PID_v1.h>
#include <PID_AutoTune_v0.h>

// Libraries for the Adafruit RGB/LCD Shield
#include <Wire.h>
#include <Adafruit_MCP23017.h>
#include <Adafruit_RGBLCDShield.h>

// Libraries for the DS18B20 Temperature Sensor
#include <OneWire.h>
#include <DallasTemperature.h>

// So we can save and retrieve settings
#include <EEPROM.h>

// ************************************************
// Pin definitions
// ************************************************

// Output Relay
#define RelayPin 7

// One-Wire Temperature Sensor
// (Use GPIO pins for power/ground to simplify the wiring)
#define ONE_WIRE_BUS 2
#define ONE_WIRE_PWR 3
#define ONE_WIRE_GND 4

// ************************************************
// PID Variables and constants
// ************************************************

//Define Variables we'll be connecting to
double Setpoint;
double Input;
double Output;

volatile long onTime = 0;

// pid tuning parameters
double Kp;
double Ki;
double Kd;

// EEPROM addresses for persisted data
const int SpAddress = 0;
const int KpAddress = 8;
const int KiAddress = 16;
const int KdAddress = 24;

//Specify the links and initial tuning parameters
PID myPID(&Input, &Output, &Setpoint, Kp, Ki, Kd, DIRECT);

// 10 second Time Proportional Output window
int WindowSize = 10000;
unsigned long windowStartTime;

// ************************************************
// Auto Tune Variables and constants
// ************************************************
byte ATuneModeRemember=2;

double aTuneStep=500;
double aTuneNoise=1;
unsigned int aTuneLookBack=20;

boolean tuning = false;

PID_ATune aTune(&Input, &Output);

// ************************************************
// DiSplay Variables and constants
// ************************************************

Adafruit_RGBLCDShield lcd = Adafruit_RGBLCDShield();
// These #defines make it easy to set the backlight color
#define RED 0x1
#define YELLOW 0x3
#define GREEN 0x2
#define TEAL 0x6
#define BLUE 0x4
#define VIOLET 0x5
#define WHITE 0x7

#define BUTTON_SHIFT BUTTON_SELECT

unsigned long lastInput = 0; // last button press
```

```
byte degree[8] = // define the degree symbol
{
 B00110,
 B01001,
 B01001,
 B00110,
 B00000,
 B00000,
 B00000,
 B00000
};

const int logInterval = 10000; // log every 10 seconds
long lastLogTime = 0;

// ************************************************
// States for state machine
// ************************************************
enum operatingState { OFF = 0, SETP, RUN, TUNE_P, TUNE_I, TUNE_D, AUTO};
operatingState opState = OFF;

// ************************************************
// Sensor Variables and constants
// Data wire is plugged into port 2 on the Arduino

// Setup a oneWire instance to communicate with any OneWire devices (not just Maxim/Dallas tempe
OneWire oneWire(ONE_WIRE_BUS);

// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

// arrays to hold device address
DeviceAddress tempSensor;

// ************************************************
// Setup and diSplay initial screen
// ************************************************
void setup()
{
   Serial.begin(9600);

   // Initialize Relay Control:

   pinMode(RelayPin, OUTPUT);    // Output mode to drive relay
   digitalWrite(RelayPin, LOW);  // make sure it is off to start

   // Set up Ground & Power for the sensor from GPIO pins

   pinMode(ONE_WIRE_GND, OUTPUT);
   digitalWrite(ONE_WIRE_GND, LOW);

   pinMode(ONE_WIRE_PWR, OUTPUT);
   digitalWrite(ONE_WIRE_PWR, HIGH);

   // Initialize LCD DiSplay 

   lcd.begin(16, 2);
   lcd.createChar(1, degree); // create degree symbol from the binary

   lcd.setBacklight(VIOLET);
   lcd.print(F("    Adafruit"));
   lcd.setCursor(0, 1);
   lcd.print(F("   Sous Vide!"));

   // Start up the DS18B20 One Wire Temperature Sensor

   sensors.begin();
   if (!sensors.getAddress(tempSensor, 0)) 
   {
      lcd.setCursor(0, 1);
      lcd.print(F("Sensor Error"));
   }
   sensors.setResolution(tempSensor, 12);
   sensors.setWaitForConversion(false);

   delay(3000);  // Splash screen

   // Initialize the PID and related variables
   LoadParameters();
   myPID.SetTunings(Kp,Ki,Kd);

   myPID.SetSampleTime(1000);
   myPID.SetOutputLimits(0, WindowSize);

  // Run timer2 interrupt every 15 ms 
  TCCR2A = 0;
  TCCR2B = 1<<CS22 | 1<<CS21 | 1<<CS20;

  //Timer2 Overflow Interrupt Enable
  TIMSK2 |= 1<<TOIE2;
}

// ************************************************
// Timer Interrupt Handler
// ************************************************
SIGNAL(TIMER2_OVF_vect) 
{
  if (opState == OFF)
  {
    digitalWrite(RelayPin, LOW);  // make sure relay is off
  }
  else
  {
    DriveOutput();
  }
```

```
}

// ************************************************
// Main Control Loop
//
// All state changes pass through here
// ************************************************
void loop()
{
   // wait for button release before changing state
   while(ReadButtons() != 0) {}

   lcd.clear();

   switch (opState)
   {
   case OFF:
      Off();
      break;
   case SETP:
      Tune_Sp();
      break;
    case RUN:
      Run();
      break;
   case TUNE_P:
      TuneP();
      break;
   case TUNE_I:
      TuneI();
      break;
   case TUNE_D:
      TuneD();
      break;
   }
}

// ************************************************
// Initial State - press RIGHT to enter setpoint
// ************************************************
void Off()
{
   myPID.SetMode(MANUAL);
   lcd.setBacklight(0);
   digitalWrite(RelayPin, LOW);  // make sure it is off
   lcd.print(F("    Adafruit"));
   lcd.setCursor(0, 1);
   lcd.print(F("   Sous Vide!"));
   uint8_t buttons = 0;

   while(!(buttons & (BUTTON_RIGHT)))
   {
      buttons = ReadButtons();
```

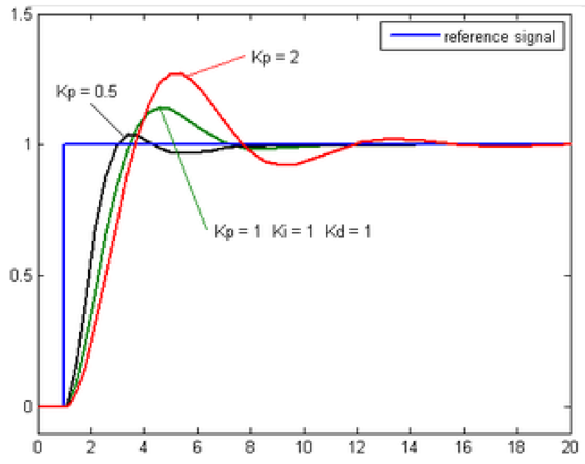
```
  }
  // Prepare to transition to the RUN state
  sensors.requestTemperatures(); // Start an asynchronous temperature reading

  //turn the PID on
  myPID.SetMode(AUTOMATIC);
  windowStartTime = millis();
  opState = RUN; // start control
}

// ************************************************
// Setpoint Entry State
// UP/DOWN to change setpoint
// RIGHT for tuning parameters
// LEFT for OFF
// SHIFT for 10x tuning
// ************************************************
void Tune_Sp()
{
  lcd.setBacklight(TEAL);
  lcd.print(F("Set Temperature:"));
  uint8_t buttons = 0;
  while(true)
  {
    buttons = ReadButtons();

    float increment = 0.1;
    if (buttons & BUTTON_SHIFT)
    {
     increment *= 10;
    }
    if (buttons & BUTTON_LEFT)
    {
      opState = RUN;
      return;
    }
    if (buttons & BUTTON_RIGHT)
    {
      opState = TUNE_P;
      return;
    }
    if (buttons & BUTTON_UP)
    {
      Setpoint += increment;
      delay(200);
    }
    if (buttons & BUTTON_DOWN)
    {
      Setpoint -= increment;
      delay(200);
    }

    if ((millis() - lastInput) > 3000)  // return to RUN after 3 seconds idle
```

```
      {
        opState = RUN;
        return;
      }
      lcd.setCursor(0,1);
      lcd.print(Setpoint);
      lcd.print(" ");
      DoControl();
    }
  }

  // ************************************************
  // Proportional Tuning State
  // UP/DOWN to change Kp
  // RIGHT for Ki
  // LEFT for setpoint
  // SHIFT for 10x tuning
  // ************************************************
  void TuneP()
  {
    lcd.setBacklight(TEAL);
    lcd.print(F("Set Kp"));

    uint8_t buttons = 0;
    while(true)
    {
      buttons = ReadButtons();

      float increment = 1.0;
      if (buttons & BUTTON_SHIFT)
      {
       increment *= 10;
      }
      if (buttons & BUTTON_LEFT)
      {
        opState = SETP;
        return;
      }
      if (buttons & BUTTON_RIGHT)
      {
        opState = TUNE_I;
        return;
      }
      if (buttons & BUTTON_UP)
      {
        Kp += increment;
        delay(200);
      }
      if (buttons & BUTTON_DOWN)
      {
        Kp -= increment;
        delay(200);
      }
      if ((millis() - lastInput) > 3000)  // return to RUN after 3 seconds idle
      {
         opState = RUN;
         return;
      }
      lcd.setCursor(0,1);
      lcd.print(Kp);
      lcd.print(" ");
      DoControl();
   }
}

// ************************************************
// Integral Tuning State
// UP/DOWN to change Ki
// RIGHT for Kd
// LEFT for Kp
// SHIFT for 10x tuning
// ************************************************
void TuneI()
{
   lcd.setBacklight(TEAL);
   lcd.print(F("Set Ki"));

   uint8_t buttons = 0;
   while(true)
   {
      buttons = ReadButtons();

      float increment = 0.01;
      if (buttons & BUTTON_SHIFT)
      {
        increment *= 10;
      }
      if (buttons & BUTTON_LEFT)
      {
         opState = TUNE_P;
         return;
      }
      if (buttons & BUTTON_RIGHT)
      {
         opState = TUNE_D;
         return;
      }
      if (buttons & BUTTON_UP)
      {
         Ki += increment;
         delay(200);
      }
      if (buttons & BUTTON_DOWN)
      {
         Ki -= increment;
         delay(200);
      }
      if ((millis() - lastInput) > 3000)  // return to RUN after 3 seconds idle
      {
         opState = RUN;
         return;
      }
      lcd.setCursor(0,1);
      lcd.print(Ki);
      lcd.print(" ");
      DoControl();
   }
}

// ************************************************
// Derivative Tuning State
// UP/DOWN to change Kd
// RIGHT for setpoint
// LEFT for Ki
// SHIFT for 10x tuning
// ************************************************
void TuneD()
{
   lcd.setBacklight(TEAL);
   lcd.print(F("Set Kd"));

   uint8_t buttons = 0;
   while(true)
   {
      buttons = ReadButtons();
      float increment = 0.01;
      if (buttons & BUTTON_SHIFT)
      {
        increment *= 10;
      }
      if (buttons & BUTTON_LEFT)
      {
         opState = TUNE_I;
         return;
      }
      if (buttons & BUTTON_RIGHT)
      {
         opState = RUN;
         return;
      }
      if (buttons & BUTTON_UP)
      {
         Kd += increment;
         delay(200);
      }
      if (buttons & BUTTON_DOWN)
      {
         Kd -= increment;
         delay(200);
      }
      if ((millis() - lastInput) > 3000)  // return to RUN after 3 seconds idle
      {
         opState = RUN;
         return;
      }
      lcd.setCursor(0,1);
      lcd.print(Kd);
      lcd.print(" ");
      DoControl();
   }
}

// ************************************************
// PID COntrol State
// SHIFT and RIGHT for autotune
// RIGHT - Setpoint
// LEFT - OFF
// ************************************************
void Run()
{
   // set up the LCD's number of rows and columns:
   lcd.print(F("Sp: "));
   lcd.print(Setpoint);
   lcd.write(1);
   lcd.print(F("C : "));

   SaveParameters();
   myPID.SetTunings(Kp,Ki,Kd);

   uint8_t buttons = 0;
   while(true)
   {
      setBacklight();  // set backlight based on state

      buttons = ReadButtons();
      if ((buttons & BUTTON_SHIFT)
         && (buttons & BUTTON_RIGHT)
         && (abs(Input - Setpoint) < 0.5))  // Should be at steady-state
      {
         StartAutoTune();
      }
      else if (buttons & BUTTON_RIGHT)
      {
        opState = SETP;
        return;
      }
      else if (buttons & BUTTON_LEFT)
      {
        opState = OFF;
        return;
      }
      DoControl();

      lcd.setCursor(0,1);
      lcd.print(Input);
      lcd.write(1);
      lcd.print(F("C : "));

      float pct = map(Output, 0, WindowSize, 0, 1000);
      lcd.setCursor(10,1);
      lcd.print(F("      "));
      lcd.setCursor(10,1);
      lcd.print(pct/10);
      //lcd.print(Output);
      lcd.print("%");

      lcd.setCursor(15,0);
      if (tuning)
      {
        lcd.print("T");
      }
      else
      {
        lcd.print(" ");
      }

      // periodically log to serial port in csv format
      if (millis() - lastLogTime > logInterval)
      {
        Serial.print(Input);
        Serial.print(",");
        Serial.println(Output);
      }

      delay(100);
   }
}

// ************************************************
// Execute the control loop
// ************************************************
void DoControl()
{
  // Read the input:
  if (sensors.isConversionAvailable(0))
  {
    Input = sensors.getTempC(tempSensor);
    sensors.requestTemperatures(); // prime the pump for the next one - but don't wait
  }

  if (tuning) // run the auto-tuner
  {
     if (aTune.Runtime()) // returns 'true' when done
     {
        FinishAutoTune();
      }
   }
   else // Execute control algorithm
   {
      myPID.Compute();
   }

   // Time Proportional relay state is updated regularly via timer interrupt.
   onTime = Output;
}

// ************************************************
// Called by ISR every 15ms to drive the output
// ************************************************
void DriveOutput()
{
  long now = millis();
  // Set the output
  // "on time" is proportional to the PID output
  if(now - windowStartTime>WindowSize)
  { //time to shift the Relay Window
     windowStartTime += WindowSize;
  }
  if((onTime > 100) && (onTime > (now - windowStartTime)))
  {
     digitalWrite(RelayPin,HIGH);
  }
  else
  {
     digitalWrite(RelayPin,LOW);
  }
}

// ************************************************
// Set Backlight based on the state of control
// ************************************************
void setBacklight()
{
   if (tuning)
   {
      lcd.setBacklight(VIOLET); // Tuning Mode
   }
   else if (abs(Input - Setpoint) > 1.0)
   {
      lcd.setBacklight(RED);  // High Alarm - off by more than 1 degree
   }
   else if (abs(Input - Setpoint) > 0.2)
   {
      lcd.setBacklight(YELLOW);  // Low Alarm - off by more than 0.2 degrees
   }
   else
   {
      lcd.setBacklight(WHITE);  // We're on target!
    }
}

// ************************************************
// Start the Auto-Tuning cycle
// ************************************************

void StartAutoTune()
{
   // REmember the mode we were in
   ATuneModeRemember = myPID.GetMode();

   // set up the auto-tune parameters
   aTune.SetNoiseBand(aTuneNoise);
   aTune.SetOutputStep(aTuneStep);
   aTune.SetLookbackSec((int)aTuneLookBack);
   tuning = true;
}

// ************************************************
// Return to normal control
// ************************************************
void FinishAutoTune()
{
   tuning = false;

   // Extract the auto-tune calculated parameters
   Kp = aTune.GetKp();
   Ki = aTune.GetKi();
   Kd = aTune.GetKd();

   // Re-tune the PID and revert to normal control mode
   myPID.SetTunings(Kp,Ki,Kd);
   myPID.SetMode(ATuneModeRemember);

   // Persist any changed parameters to EEPROM
   SaveParameters();
}

// ************************************************
// Check buttons and time-stamp the last press
// ************************************************
uint8_t ReadButtons()
{
  uint8_t buttons = lcd.readButtons();
  if (buttons != 0)
  {
    lastInput = millis();
  }
  return buttons;
}
```

```
// ************************************************
// Save any parameter changes to EEPROM
// ************************************************
void SaveParameters()
{
   if (Setpoint != EEPROM_readDouble(SpAddress))
   {
      EEPROM_writeDouble(SpAddress, Setpoint);
   }
   if (Kp != EEPROM_readDouble(KpAddress))
   {
      EEPROM_writeDouble(KpAddress, Kp);
   }
   if (Ki != EEPROM_readDouble(KiAddress))
   {
      EEPROM_writeDouble(KiAddress, Ki);
   }
   if (Kd != EEPROM_readDouble(KdAddress))
   {
      EEPROM_writeDouble(KdAddress, Kd);
   }
}

// ************************************************
// Load parameters from EEPROM
// ************************************************
void LoadParameters()
{
  // Load from EEPROM
   Setpoint = EEPROM_readDouble(SpAddress);
   Kp = EEPROM_readDouble(KpAddress);
   Ki = EEPROM_readDouble(KiAddress);
   Kd = EEPROM_readDouble(KdAddress);

   // Use defaults if EEPROM values are invalid
   if (isnan(Setpoint))
   {
     Setpoint = 60;
   }
   if (isnan(Kp))
   {
     Kp = 850;
   }
   if (isnan(Ki))
   {
     Ki = 0.5;
   }
   if (isnan(Kd))
   {
     Kd = 0.1;
   }
}
```

```
// ************************************************
// Write floating point values to EEPROM
// ************************************************
void EEPROM_writeDouble(int address, double value)
{
  byte* p = (byte*)(void*)&value;
  for (int i = 0; i < sizeof(value); i++)
  {
    EEPROM.write(address++, *p++);
  }
}

// ************************************************
// Read floating point values from EEPROM
// ************************************************
double EEPROM_readDouble(int address)
{
  double value = 0.0;
  byte* p = (byte*)(void*)&value;
  for (int i = 0; i < sizeof(value); i++)
  {
    *p++ = EEPROM.read(address++);
  }
  return value;
}
```

Tuning



> *Diagram from Wikipedia entry: PID Controller (http://adafru.it/ceV)*

Default Tuning

The default tuning parameters in the controller sketch are average values from a few different models of rice cooker. But there will be variations even between two cookers of the same model from the same manufacturer.

Auto Tuning

The Auto tuning function of the sketch can determine 'ballpark' parameters for your cooker. You can refine the tuning from there.

To use the Autotune function, first let the cooker pre-heat and reach setpoint temperature. Auto-tuning works best if the system is already at a steady-state. The "Run" state will not allow you to invoke the auto-tuner, unless you are already within 0.5 degrees of the setpoint temperature.

Once the cooker has stabilized at or near the setpoint, press the **RIGHT** and **SELECT** buttons simultaneously. The backlight will turn violet to indicate that you are in autotune mode.

Be patient while the system tunes itself. It typically will take an hour or more for the auto-tune algorithm to complete. When the auto-tuning is complete, the backlight will return to the normal operating color scheme. The auto-tuned parameters will be saved to EEPROM, so they are ready for the next time you use it.

Manual Tuning

As mentioned before, auto-tuning is not perfect. With some practice you can probably get closer to the ideal tuning for your cooker. There are many good resources on the web that explain PID tuning. Here are just a couple:

PID Controller Tuning: A Short Tutorial (http://adafru.it/ceW)

Simple PID tuning rules (http://adafru.it/ceX)

To tune the Kp, Ki and Kd parameters, use the RIGHT button to navigate between the tuning screens. The UP and DOWN buttons will change the value. Pressing SELECT while also pressing UP or DOWN will change the value 10x faster.

Manual Tuning Hint

One thing to be aware of is that the temperature control on a rice cooker is *non-linear* and *asymmetrical*. You can apply heat, but there is no active cooling. As a result, most rice cookers take a long time to recover from a temperature *overshoot*. For this reason, it is usually better to aim for a slightly *overdamped* response to avoid overshoot.

Cook with it!

The best part of making your own sous vide setup is the testing portion, yum!

Sous vide uses lower than normal cooking temperatures. If not done with care, this can create conditions that promote the growth of harmful bacteria. For an excellent guide to safe sous vide cooking temperatures and food handling practices, as well as time and temperature charts, see Douglas Baldwin's book and web-site "A Practical Guide to Sous Vide Cooking (http://adafru.it/ceY)"

Cook a 'perfect' egg!

A good first test for your new cooker is a 'perfect egg'. Eggs require no special preparation for sous vide cooking. And, since they are very sensitive to small variations in cooking temperature, they are a good test of the accuracy of your temperature controller.

Different chefs have different ideas on exactly what constitutes a perfect egg. But the precise temperature control of your sous vide cooker will let you achieve your perfect egg every time.

http://www.edinformatics.com/math_science/science_of_cooking/eggs_sous_vide.htm (http://adafru.it/ceZ)

Cook a steak!

As with an egg, the precise temperature control of your cooker will allow you to cook steaks to the right level of doneness with 100% repeatability. No more overdone edges and raw in the middle. Your steak will will be cooked the same from edge to edge. Brown them on the grill for just few seconds on each side before serving.

> > > > *Sirloin Tip-Strip with Grilled Zuccini*
> > > > *Cooked 90 minutes @ 57C.*

Cook a Fish!

Fish can be tricky to cook. Just a few seconds too long in the skillet and goes from moist and flakey to dry and crumbly. With sous vide, you can be sure that it is cooked through, but not overdone.

Haddock Fillets with haricots verts and orange saffron sauce. (one of our favorites!)
Cooked 20 minutes @ 54.5C

Downloads and Links

Sous Viduino Arduino Code

You can get the latest code from the github repository at
https://github.com/adafruit/Sous_Viduino (http://adafru.it/ceU)

Library Downloads:

Arduino PID Library (http://adafru.it/ceR)
PID Autotune Library (http://adafru.it/cf0)
You will also need the libraries for the DS18B20 temperature sensor and RGB LCD shields, check the product & kit pages for details on those items.

There are several versions of the DS18B20 library out there. The code for this guide was compiled with version 3.7.2 which can be downloaded here:
Dallas Temperature Lbrary (http://adafru.it/ejf)

And the PJRC version of the OneWire library

OneWire Library (http://adafru.it/das)

Library Documentation:

PID Algorithm Documentation (http://adafru.it/ceS)
PID Autotune Documentation (http://adafru.it/ceT)

Additional information about sous vide cooking

Douglas Baldwin's "A Practical Guide to Sous Vide Cooking (http://adafru.it/ceY)"

For Leonardo Users:

Leonardo Timer DIfferences

Customer and forum member ytoff57 (http://adafru.it/aMD) (http://adafru.it/aMD)has successfully ported this code to the Leonardo. One difference between the Leonardo and the Arduino is the timers, so the timer interrupt code does not work on that platform. His has tested the following modifications based on the TimerOne library:

```
#include <TimerOne.h>
//...
void setup()
{
  //...
  Timer1.initialize(15000);
  Timer1.attachInterrupt(TimerInterrupt);
}
//..
void TimerInterrupt()
{
  if (opState == OFF)
  {
    digitalWrite(RelayPin, LOW);  // make sure relay is off
  }
  else
  {
    DriveOutput();
  }
}
```