:: Thermistor4 Universal Library For Temperature Calculations ::

Navigation: History Schematic Header Library Sketch

Thermistor4 differs from the previous thermistor incarnations in that it is designed to be fully universal using an NTC thermistor with the full Steinhart-Hart equations.

- Brief instructions are given in the source code to use http://thermistor.sourceforge.net to accurately set up an unknown NTC thermistor.
- This library is written in C++ for object oriented flexibilty. Since the variables are fully contained in the object, multiple thermistors of different types and wirings may be freely intermixed.
- No look-up tables are used or generated. Everything is equation driven.
- For full and working Steinhart-Hart coefficients, simplified, standard, and extended equations may be chosen for processing time and accuracy.
- An "Offset" variable is included for tweaking the final calibration.
- It can use an internal or external ADC (any the AVR can communicate with). The external ADC will need some small glue code to fetch the reading. Instructions are in the source code. If an all-in-one temperature sensor returns a number for use with Steinhart-Hart equations, it may also be used.
- The bit resolution of the ADC doesn't matter. Most ATmega's come with 10bit native, but lower and higher bit resolutions are accounted for.
- Input voltage can be different for each thermistor if needed. Since it is a normal variable, it can be changed on the fly for use in something like battery powered systems. Note that the input voltage to the ADC pin must still be within its operating range.
- The resistor divider can be of any value. Brief instructions and an equation are given in the source code to help optimize the value.
- Output temperatures are in the form of Kelvin, centigrade, and Farenheit.
- This uses the new floating point Serial.print() in Arduino 0018. Older Arduino versions could be used, but the reporting code will need to be modified accordingly.
- The source code is excessively commented to help with my memory problems. You can take advantage of that as a learning experience of what to do and not to do.

On that note, PLEASE read the source code carefully. It was designed to be both instructional and how to properly set things up for usage. Once set up and calibrated, I recommend disabling the debugging code for size reasons.

Each section below is designed to be copied and pasted out into the file name of the title header (watch out if there's word wrap). To run the example code, create a "Thermistor4c" subdirectory inside your sketch directory and copy in the three files. The example code prints out a lot and is recommended for initial testing and calibration.

Note that my Steinhart-Hart coefficients for my two thermistors will likely NOT work for yours.

History

- 2010-10-17 Version 4c: Added sizeof() to Thermistor4SerialPrint() to get an

idea of space requirements. Added pull-up loop in setup() to avoid floating pins.

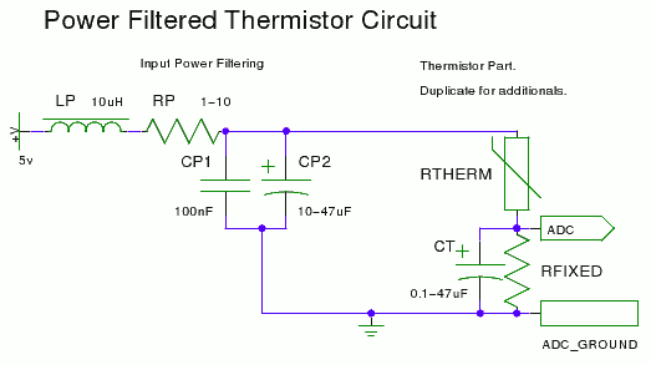
- 2010-09-07 Version 4b: Moved ReadCalculate() 1-2-3 levels example into Thermistor4SerialPrint() for permanency. Added bottom/middle/top ADC resolution check to Thermistor4SerialPrint(). Added explicit "this->" usage in class. Compiled code size and runtimes are unchanged, but allows for better clarity and less confusion if global variables have the same name as class variables. Changed "float" to "double". Checked and avr-gcc has both using 4 bytes. This might be a little more accurate if this library is ported to a PC and the Arduino is used as an external ADC.

- 2010-09-04 Version 4a: Initial release version. Fixed some spelling errors, cleaned up code to look a little better in the wiki, updated the schematic, and added the averaging example code.

I don't have an external ADC connected to my "2009" board and would appreaciate some help testing that.

Schematic

Todo: This image shouldn't be rescaled up like it is.



The "Input Power Filtering" section doesn't have to be that elaborate, but will give cleaner results. The inductor (LP) and the RC filter (RP+CP1+CP2) both filter out high frequency noise. CP2 will also help stabilize any voltage fluctuations. If the rails are clean enough, most people will just do CP1 and/or CP2.

The thermistor block can be duplicated for multiple thermistors. RTHERM is the thermistor. RFIXED corresponds to ResistanceFixed in the code and is calculated based on RTHERM and the desired range. CT is a stabilizing capacitor that will help remove jitter. Thermistors aren't very fast by nature, so even at upper values, the readings will still be reasonably fast.

For current to flow, power grounds must be connected between the power supply and the ADC. If the Arduino is powering the ADC and thermistor, this is already taken care of.

To power another thermistor block, take its power right after CP2.

Whatever ADC is used is connected to the ADC pin (obviously).

Thermistor4.h

```
// Thermistor4.h definition of class. For Arduino 0018 or newer.
// Thermistor class to consolidate various variables and functions.
// (c)20101017 by MODAT7 under GNU GPLv1

/*
Purpose: To have higher accuracy readings from various sized negative temperature coefficient
(NTC) thermistors of varying resistances using the Steinhart-Hart equations. If all that is
needed is relative temperature changes from a fixed point, a simpler model is recommended rather
than using this large class. On the flip side, heavier code is often better than having a number
of large look up tables for a number of different thermistors.

In plain English: It's nice to re-use thermistors from dead indoor/outdoor thermometers in
projects. It's also nice to recalibrate thermistors that are "supposed" to be calibrated.
At this date, I haven't seen a consolidated project/class library that does all this. Some will
```

say there are ways to make this code a bit cleaner and more efficient. It's free so, have at it. The purpose of this coding style is to be clear and easy to read where many have been so murky.

To make this library universally permanent from the "Sketch / Import Library" menu, go into the Arduino install directory, "libraries" subdirectory, and create a "Thermistor4" directory. Copy in Thermistor4.h and Thermistor4.cpp. Restart the Arduino IDE if it is running.

To use this library in a single project, exit the Arduino IDE if it is running and copy in Thermistor4.h and Thermistor4.cpp into the project directory. It will show up in the extra tabs when the IDE is relaunched. Add a #include "Thermistor4.h" into the sketch.

A C++ class makes multiple thermistors easy to handle. In the current form, it is also easy to change certain variables on the fly to keep accuracy (such as if VoltageSupply is a battery that lowers over time). When setting up and calibrating, a serial routine could be written to tweak on the fly.

Provisions have been made for external ADC's (serial or parallel, whatever Arduino can interface with), but you need to program how the raw data gets to this class object yourself. See the ReadADC() comments for instructions.

Potential usages. I use mine for non-critical medical and high power component monitoring (power transistors and MOSFET's). Others include indoor/outdoor thermometer weather stations, indoor AC zone vent control and monitoring (if using home automation), AC thermostats (like for uncontrolled window units), controlling attic ventilation fans (home automation with relays), garden greenhouse monitoring, refridgerator and freezer monitoring, cooking (*IF* using a food grade probe that is rated for those temperatures), computer hardware temperature monitoring (case zones and chips, maybe add fan controls), and probably many more...

NOTE: Read this and the comments carefully for usage. It isn't very hard but there is A LOT OF TEDIUM involved in getting this to work correctly.

NOTE: To get the best calibration out of thermistor.sf.net, take at least several evenly spaced readings and go 5-10% beyond the lowest and highest desired measuring points. I personally use the freezer (for below freezing), ice water (for freezing), multiple indoor and outdoor temps (against a glass vial type thermometer), and boiling water.

NOTE: The thermistor.sf.net calculations do NOT seem to produce 100% correct coefficients for me. The full calculation (CalculateTemperature(3)) seems to work very well, but any of the shorter equations produce garbage. Check these carefully in your project before usage.

As just mentioned ReadCalculate() and CalculateTemperature() have 3 different levels for the 3 different complexities of the equation. 1 is the fastest/least accurate, and 3 is the slowest/most accurate. Using another example, I figure the fastest/1 is about 2-3 degrees Kelvin off from the most accurate.

Choosing "ResistanceFixed". The following equation can be helpful in getting the maximum voltage swing out of a variable resistor for largest accuracy range.
Vmax = (VoltageSupply * ResistorFixed) / (Thermistor_Min + ResistorFixed)
"Vmax" is what the ADC pin would see when resistance is at "Thermistor_Min". This equation could also be used to narrow down resolution to a particular range of interest.

Long wires on the thermistor could also contribute to a mild voltage loss. VoltageSupply may need to be dropped a little.

If multiple thermistors are being used, pick one that is most likely to be the most accurate and put all the other thermistors close to it. Let them all sit awhile and adjust to room temperature. Once they stabilize, tweak the thermistor's "Offset" value until they all match the most accurate one.

Some have mentioned that thermistors will self heat and skew the reading. This is true for lower resistance thermistors and less of a problem for higher resistance ones (more resistance is less current is less heat). The equations in this class do not account for self generated heat.
For any unused ADC input pins, it's best to set their pullup resistors so they don't mess with the others (pinMode(myPin, INPUT); digitalWrite(myPin, HIGH)).

Sources: This came from all over but mainly:
http://thermistor.sourceforge.net
   (thanks to SoftQuadrat, good explanation, use the coeff/simu.txt calibration tools here)
http://en.wikipedia.org/wiki/Thermistor
http://en.wikipedia.org/wiki/Steinhart-Hart_equation
Arduino: ComponentLib/Thermistor2 and some forum discussions.

Observation. While the math for all this is well within an ATmega's capability, it is still rather heavy for an MCU. If the Arduino is plugged into a PC and since this library is written in C++, it wouldn't be hard to port the bulk of it over to the PC side. The Arduino hardware would provide the raw ADC numbers to the PC, and the PC would do the calculations. The PC version of the library would treat the Arduino hardware like an external ADC.

Physical Circuit:
VoltageSupply---Thermistor---\ADC Pin/---Fixed Resistor---Ground

VoltageSupply needs to be very clean and stable for this to be accurate (especially during ADC reading). Adding voltage regulation/smoothing and stabilizing capacitors between VoltageSupply and Ground may help. Adding a small capacitor (low micro-Farad range) between the ADC pin and ground may also help. Too large a capacitor will slow the meter readings down but should still work. Larger capacitors will help smooth the jitter out of multi-hour graphs.

Code Usage:

```
Global:
//Inside thermistor on Arduino ADC pin 0, Outside on pin 1.
#define THERMISTORPinInside 0
#define THERMISTORPinOutside 1
//One temperature monitoring thermistor for Inside and Outside.
Thermistor4 ThermistorInside, ThermistorOutside;
unsigned long ThermistorLastMillis; //last time something was run.
//If using protothreads...
static struct pt ptv_ThermistorReport; //these hold the states of the PT's.

setup():
//My 2 salvaged thermistors are about 30k at room temperature.
ThermistorInside.Pin = THERMISTORPinInside; //Set the pin number.
ThermistorOutside.Pin = THERMISTORPinOutside;
ThermistorInside.SetUp(); //Sets up the analog read pin for internal AVR.
ThermistorOutside.SetUp(); //If using an external ADC, write your own setup here.
//pow() is used elsewhere so might as well be used here.
ThermistorInside.BitResolution=pow(2, 10)-1; //ATmega's have a 10bit ADC (2^10-1).
ThermistorOutside.BitResolution=pow(2, 10)-1; //An external ADC may be lower or higher than 10bits.
ThermistorInside.VoltageSupply=4.95; //My USB powers my ATmega325 chip at 4.95v. Meter this for accuracy.
ThermistorOutside.VoltageSupply=4.95; //An external ADC may have different voltages. Meter this for accuracy.
ThermistorInside.ResistanceFixed=27200; //Fixed resistor in the divider. Measured in ohms. Meter this for acc
ThermistorOutside.ResistanceFixed=27100; //The resistor divider should be calculated to maximize desired rang
ThermistorInside.Offset=0.5; //adjust temperature in Kelvin up or down a little to account for unforseen vari
ThermistorOutside.Offset=0.5; //This will be by trial and error during final manual calibration.
//These numbers were generated from thermistor.sf.net and aren't quite right unless using the full equation.
ThermistorInside.SteinhartA1=1.560442157476244e-003;  //First Steinhart-Hart coefficient.
ThermistorInside.SteinhartA2=-1.298742723052728e-005; //Second Steinhart-Hart coefficient.
ThermistorInside.SteinhartA3=2.500508035979886e-005;  //Third Steinhart-Hart coefficient.
ThermistorInside.SteinhartA4=-7.698170259653937e-007; //Fourth Steinhart-Hart coefficient.
ThermistorOutside.SteinhartA1=2.975623989921602e-003;  //First Steinhart-Hart coefficient.
ThermistorOutside.SteinhartA2=-4.448067018378571e-004; //Second Steinhart-Hart coefficient.
ThermistorOutside.SteinhartA3=6.848365975770642e-005;  //Third Steinhart-Hart coefficient.
ThermistorOutside.SteinhartA4=-2.217688558250577e-006; //Fourth Steinhart-Hart coefficient.
//If using proto-threads, PT's need initializing before use.
PT_INIT(&ptv_ThermistorReport);

loop():
//PROTOTHREAD VERSION:
//loop forever and let the PT scheduler decide what needs running.
//The PT's will not run until their functions are called.
//If the functions exit, being in the loop will restart the thread.
//Each thread gets its own "pt" state variable.
//Since arguments and local function variables can't be used, use the class object's.
ptf_ThermistorReport(&ptv_ThermistorReport);
//This function exists somewhere else and acts on the thermistor values.

//NORMAL VERSION:
ThermistorInside.ReadCalculate(3);
Serial.print("Inside Temp: "); Serial.println(ThermistorInside.GetFarenheit(), 2);
ThermistorOutside.ReadCalculate(3);
Serial.print("Outside Temp: "); Serial.println(ThermistorOutside.GetFarenheit(), 2);

//Example CSV output that can be extracted from the other information and plotted.
//unix/linux: grep "^CSV," > outfile.csv
ThermistorInside.ReadCalculate(3);
Serial.print("CSV,"); Serial.print(ThermistorInside.GetFarenheit(), 2);
Serial.print(","); Serial.print(ThermistorInside.GetCentigrade(), 2);
ThermistorOutside.ReadCalculate(3);
Serial.print(","); Serial.print(ThermistorOutside.GetFarenheit(), 2);
Serial.print(","); Serial.println(ThermistorOutside.GetCentigrade(), 2);

*/

#ifndef THERMISTOR4_H
#define THERMISTOR4_H

//Bring in the Arduino stuff since it gets called.
#include <WProgram.h>
//for log() (natural logarithm, not log10).
#include <math.h>

//Set to 1 to include debug code, set to 0 to exclude.
```

```
//Extra prints add about 1k to the binary.
//This is convenient for calibrating, but usually unnecessary for a final project.
#define THERMISTORDEBUG 1

// Thermistor class to consolidate various variables and functions.
class Thermistor4 {

public:
//note: floats and doubles are both 4 bytes in avr-gcc.
unsigned char Pin; // analog pin number on the Arduino board the thermistor circuit is connected to.
unsigned int BitResolution; // such as an 8bit, 10bit, or 12bit ADC. Most newer Arduino boards are 10bit.
unsigned int ADCReading; // last reading from ADC.
double VoltageSupply; // supply voltage of the thermistor-divider. Manually meter this for accuracy.
double VoltageReading; // current voltage in the middle point of the thermistor-divider.
unsigned long ResistanceFixed; // fixed resistor between thermistor and ground, measured in ohms. Meter this
unsigned long ResistanceThermistor; // last calculated resistance of the thermistor, measured in ohms.
double SteinhartA1; // Calibrate/recalibrate the thermistor from: http://thermistor.sourceforge.net
double SteinhartA2; // SF and real SH coefficients are somehow different.
double SteinhartA3; // These 4 Steinhart-Hart coefficients are different for every thermistor...
double SteinhartA4; // ...and will help make sure the thermistor calculations are accurate over a wide range.
double Temperature; // calculated temperature in Kelvin.
double Offset; // adjust temperature in Kelvin up or down a little to account for unforseen variations.

//Thermistor4(); //Constructor, removed for size.
void SetUp();
void ReadADC(unsigned int);
void CalculateTemperature(unsigned char);
void ReadCalculate(unsigned char);
double GetCentigrade();
double GetFarenheit();
#if THERMISTORDEBUG
void Thermistor4SerialPrint();
#endif
};

#endif

//end of Thermistor4.h
```

Thermistor4.cpp

```
// Thermistor4.cpp member functions of class. For Arduino 0018 or newer.
// Thermistor class to consolidate various variables and functions.
// (c)20101017 by MODAT7 under GNU GPLv1

#include "Thermistor4.h"

//The classes would normally use read() and set() commands and "private" the data,
//but space is at a premium here in an MCU. This has been written for flexibility.
//"this->" allows for better clarity and less confusion if global variables have
//the same name as class variables.

//constructor. This could be more but I want to code for size small.
/* Space is at a premium and this isn't necessary since it is taken care of in setup().
Thermistor4::Thermistor4() {
  this->Pin=0;
  this->BitResolution=10;
  this->SteinhartA1=0.0;
  this->SteinhartA2=0.0;
  this->SteinhartA3=0.0;
  this->SteinhartA4=0.0;
  this->Offset=0.0;
}
*/

//Set up and make sure the Arduino analog pin is in the proper mode.
//Use your own custom SetUp() for an external ADC.
void Thermistor4::SetUp() {
//Take the pin out of pull-up mode.
//Arduino analog pins are 14-19 when accessed in digital mode.
pinMode((14 + this->Pin), INPUT);
digitalWrite((14 + this->Pin), LOW);
//The first analog read of a pin may be bad, so read and discard it.
analogRead(this->Pin);
}

//Read the raw value of the ADC and calculate stuff needed for Steinhart-Hart equation.
//ExternalReading is 0 for the Arduino's internal ADC's. Reading 0 on an ADC
//thermistor is a very unlikely condition.
```

```
//For an external ADC: ExternalReading is whatever the external ADC's return number is.
//You will have to write your own code to get this and call ReadADC() and
//CalculateTemperature() afterwards instead of the usual ReadCalculate().
void Thermistor4::ReadADC(unsigned int ExternalReading) {
if(ExternalReading == 0) {
  this->ADCReading = analogRead(this->Pin); //takes about 100uS to read an analog Arduino pin.
  }
else {this->ADCReading = ExternalReading;} //set to the provided external ADC value.
//Calculate the current voltage at Pin.
//Voltage Divider Equation: VMidPoint = (RBottom / (RTop+RBottom)) * VSupply
//...solving for: RTop = ((VSupply*Rbottom)/VMidPoint) - RBottom
this->VoltageReading = ((double)this->ADCReading/(double)this->BitResolution) * this->VoltageSupply;
this->ResistanceThermistor = ((this->VoltageSupply*(double)this->ResistanceFixed) / this->VoltageReading) - th
} //end ReadADC()


/*
//Calculate the ADC reading into temperature in Kelvin. (from http://thermistor.sf.net method)
//I don't know why these look so different than the usual Steinhart-Hart equations.
//These look backwards?
//This code is left here for legacy reasons and as an alternative example.
//It is actually about 5% faster if always doing the full equations.
void Thermistor4::CalculateTemperature() {
double LnResist;
LnResist = log(this->ResistanceThermistor); // no reason to calculate this multiple times.
//Build up the number in the stages.
this->Temperature = this->SteinhartA4;
this->Temperature = this->Temperature * LnResist + this->SteinhartA3;
this->Temperature = this->Temperature * LnResist + this->SteinhartA2;
this->Temperature = this->Temperature * LnResist + this->SteinhartA1;
//Final part is to invert
this->Temperature = (1.0 / this->Temperature) + this->Offset;
} //end CalculateTemperature()
*/


//Calculate the ADC reading into temperature in Kelvin. (official Steinhart-Hart equations)
//This is split up (Selection) to allow for comparisons and easy selection of the
//Steinhart-Hart complexity levels.
//Steinhart-Hart Thermistor Equations (in Kelvin) Selections:
//1: Simplified: Temp = 1 / ( A + B(ln(R)) )
//2: Standard:   Temp = 1 / ( A + B(ln(R)) + D(ln(R)^3) )
//3: Extended:   Temp = 1 / ( A + B(ln(R)) + C(ln(R)^2) + D(ln(R)^3) )
//Obviously lower numbers are less accurate but are much faster.
void Thermistor4::CalculateTemperature(unsigned char Selection) {
double LnResist;
LnResist = log(this->ResistanceThermistor); // no reason to calculate this multiple times.
//build up the Steinhart-Hart equation depending on Selection.
//Level 1 is used by all.
this->Temperature = this->SteinhartA1 + (this->SteinhartA2 * LnResist);
//If 2, add in level 2.
if(Selection>=2) { this->Temperature = this->Temperature + (this->SteinhartA4 * LnResist * LnResist * LnResist
//If 3, add in level 3.
if(Selection>=3) { this->Temperature = this->Temperature + (this->SteinhartA3 * LnResist * LnResist); }
//Final part is to invert
this->Temperature = (1.0 / this->Temperature) + this->Offset;
} //end CalculateTemperature()


//Read and calculate temperature. This combines ReadADC() and CalculateTemperature() for
//convenience. This one is the one that will generally be used. Do not use this if using an
//external ADC.
void Thermistor4::ReadCalculate(unsigned char Selection) {
this->ReadADC(0); // 0 means use internal ADC.
this->CalculateTemperature(Selection);
} //end ReadCalculate()


/* Note:
Centigrade and Farenheit aren't stored internally for size reasons. In most cases these functions
won't be called more than once per reading, so this is a non-issue. If they would be called more
than once, create an external double variable and assign the value to it.
*/


//Return the temperature in Centigrade.
//Since many will want one system or the other, this is split apart.
double Thermistor4::GetCentigrade() {
return this->Temperature - 273.15;  // from Kelvin to Centigrade
} //end GetCentigrade()
```

```
//Return the temperature in Farenheit.
//Since many will want one system or the other, this is split apart.
double Thermistor4::GetFarenheit() {
return (this->Temperature - 273.15) * (9.0 / 5.0) + 32.0; // from Kelvin to Centigrade to Farenheit
} //end GetFarenheit()


#if THERMISTORDEBUG
//Print out the vaule of all the current variables. Mainly for debugging.
//If needed, this function can be disabled to save space after calibration and setup.
//Extra code adds about 2.5k to the binary.
void Thermistor4::Thermistor4SerialPrint() {
  double mytempc, mytempf;
  //print out this class's variables.
  this->ReadCalculate(3);
  Serial.println("Thermistor4 Values:");
  Serial.print("Pin: "); Serial.println(this->Pin, DEC);
  Serial.print("BitResolution: "); Serial.println(this->BitResolution, DEC);
  Serial.print("ADCReading: "); Serial.println(this->ADCReading, DEC);
  Serial.print("VoltageSupply: "); Serial.println(this->VoltageSupply, DEC);
  Serial.print("VoltageReading: "); Serial.println(this->VoltageReading, DEC  );
  Serial.print("ResistanceFixed: "); Serial.println(this->ResistanceFixed, DEC);
  Serial.print("ResistanceThermistor: "); Serial.println(this->ResistanceThermistor, DEC);
  //it would be nice if Serial.print() would do doubles as powers...
  Serial.print("SteinhartA1*1,000,000: "); Serial.println((this->SteinhartA1*1000000), DEC);
  Serial.print("SteinhartA2*1,000,000: "); Serial.println((this->SteinhartA2*1000000), DEC);
  Serial.print("SteinhartA3*1,000,000: "); Serial.println((this->SteinhartA3*1000000), DEC);
  Serial.print("SteinhartA4*1,000,000: "); Serial.println((this->SteinhartA4*1000000), DEC);
  Serial.print("Temperature Offset: "); Serial.println(this->Offset, DEC);
  Serial.print("Temperature Kelvin: "); Serial.println(this->Temperature, DEC);
  Serial.print("Temperature Centigrade: "); Serial.println(this->GetCentigrade(), DEC);
  Serial.print("Temperature Farenheit: "); Serial.println(this->GetFarenheit(), DEC);
  Serial.print("sizeof(Thermistor4) object: "); Serial.println(sizeof(Thermistor4),DEC);
  //Show differences between ReadCalculate levels.
  //If the numbers are very different, the coefficients aren't quite right. This seems common
  //for self generated numbers from http://thermistor.sf.net. Use the full version (3) instead
  //of the shorter ones.
  Serial.println("If the first 2 temps are nonsense, then the Steinhart-Hart coefficients aren't quite right.
  Serial.println("You will have to use ReadCalculate(3) for all the temperature readings.");
  this->ReadCalculate(1);
  Serial.print("Inside Farenheit accuracy, ReadCalculate 1, 2, 3: ");
  Serial.print(this->GetFarenheit(), DEC);
  this->CalculateTemperature(2);
  Serial.print(", ");
  Serial.print(this->GetFarenheit(), DEC);
  this->CalculateTemperature(3);
  Serial.print(", ");
  Serial.println(this->GetFarenheit(), DEC);
  //Check ADC resolution/calculation throughout the range.
  //Some will be linear (voltage), others will not (resistance).
  //This is important to know for tight tolerance values.
  //Check the resolution at the bottom of the range.
  this->ReadADC(this->BitResolution/10);
  this->CalculateTemperature(3);
  mytempc=this->Temperature;
  mytempf=this->GetFarenheit();
  this->ReadADC((this->BitResolution/10)+1);
  this->CalculateTemperature(3);
  Serial.print("Temperature Resolution ADC Bottom (K/C, F): ");
  Serial.print((this->Temperature-mytempc), DEC);
  Serial.print(", ");
  Serial.println((this->GetFarenheit()-mytempf), DEC);
  //Check the resolution at the middle of the range.
  this->ReadADC(this->BitResolution/2);
  this->CalculateTemperature(3);
  mytempc=this->Temperature;
  mytempf=this->GetFarenheit();
  this->ReadADC((this->BitResolution/2)+1);
  this->CalculateTemperature(3);
  Serial.print("Temperature Resolution ADC Middle (K/C, F): ");
  Serial.print((this->Temperature-mytempc), DEC);
  Serial.print(", ");
  Serial.println((this->GetFarenheit()-mytempf), DEC);
  //Check the resolution at the top of the range.
  this->ReadADC(this->BitResolution*9/10);
  this->CalculateTemperature(3);
  mytempc=this->Temperature;
  mytempf=this->GetFarenheit();
  this->ReadADC((this->BitResolution*9/10)+1);
  this->CalculateTemperature(3);
```

```
  Serial.print("Temperature Resolution ADC Top (K/C, F): ");
  Serial.print((this->Temperature-mytempc), DEC);
  Serial.print(", ");
  Serial.println((this->GetFarenheit()-mytempf), DEC);
} //end Thermistor4SerialPrint()
#endif

//end of Thermistor4() class.
```

Thermistor4c.pde

```
// Thermister4 Library version 4c. For Arduino 0018 or newer.
// (c)20101017 by MODAT7 under GNU GPLv1

//Example code of various functions using my 2 salvaged thermistors.

#include "Thermistor4.h"

//Faster serial means shorter delays in the program.
#define SERIALBAUD 115200

//Inside thermistor on Arduino ADC pin 0, Outside on pin 1.
#define THERMISTORPinInside 0
#define THERMISTORPinOutside 1

//======================================

//One temperature monitoring thermistor for Inside and Outside.
Thermistor4 ThermistorInside, ThermistorOutside;
//various temp variables for testing.
unsigned int i, ADCAverage;
double mytemp;
unsigned long mytime1, mytime2; //for speed tests


//======================================
void setup() {

//pin floating paranoia, set pullup's (don't mess with 0 and 1 serial).
for(int i=2; i<20; i++) {
  pinMode(i, INPUT);
  digitalWrite(i, HIGH);
  }

Serial.begin(SERIALBAUD); //set up serial port.

//My 2 salvaged thermistors are about 30k at room temperature.
ThermistorInside.Pin = THERMISTORPinInside; //Set the pin number.
ThermistorOutside.Pin = THERMISTORPinOutside;
ThermistorInside.SetUp(); //Sets up the analog read pin for internal AVR.
ThermistorOutside.SetUp(); //If using an external ADC, write your own setup here.
//pow() is used elsewhere so might as well be used here.
ThermistorInside.BitResolution=pow(2, 10)-1; //ATmega's have a 10bit ADC (2^10-1).
ThermistorOutside.BitResolution=pow(2, 10)-1; //An external ADC may be lower or higher than 10bits.
ThermistorInside.VoltageSupply=4.95; //My USB powers my ATmega325 chip at 4.95v. Meter this for accuracy.
ThermistorOutside.VoltageSupply=4.95; //An external ADC may have different voltages. Meter this for accuracy.
ThermistorInside.ResistanceFixed=27200; //Fixed resistor in the divider. Measured in ohms. Meter this for acc
ThermistorOutside.ResistanceFixed=27100; //The resistor divider should be calculated to maximize desired rang
ThermistorInside.Offset=0.4; //adjust temperature in Kelvin up or down a little to account for unforseen vari
ThermistorOutside.Offset=0.4; //This will be by trial and error during final manual calibration.
//These numbers were generated from thermistor.sf.net and aren't quite right unless using the full equation.
ThermistorInside.SteinhartA1=1.560442157476244e-003;  //First Steinhart-Hart coefficient.
ThermistorInside.SteinhartA2=-1.298742723052728e-005; //Second Steinhart-Hart coefficient.
ThermistorInside.SteinhartA3=2.500508035979886e-005;  //Third Steinhart-Hart coefficient.
ThermistorInside.SteinhartA4=-7.698170259653937e-007; //Fourth Steinhart-Hart coefficient.
ThermistorOutside.SteinhartA1=2.975623989921602e-003;  //First Steinhart-Hart coefficient.
ThermistorOutside.SteinhartA2=-4.448067018378571e-004; //Second Steinhart-Hart coefficient.
ThermistorOutside.SteinhartA3=6.848365975770642e-005;  //Third Steinhart-Hart coefficient.
ThermistorOutside.SteinhartA4=-2.217688558250577e-006; //Fourth Steinhart-Hart coefficient.

} //end setup()


//======================================
void loop() {

#if THERMISTORDEBUG
//Show the debugging information:
Serial.println("");
```

```
Serial.println("ThermistorInside Debug: ");
ThermistorInside.Thermistor4SerialPrint();
Serial.println("");
Serial.println("ThermistorOutside Debug: ");
ThermistorOutside.Thermistor4SerialPrint();
Serial.println("");
#endif

//Nice printout form.
ThermistorInside.ReadCalculate(3);
Serial.print("Inside Temp (F): "); Serial.println(ThermistorInside.GetFarenheit(), 2);
ThermistorOutside.ReadCalculate(3);
Serial.print("Outside Temp (F): "); Serial.println(ThermistorOutside.GetFarenheit(), 2);

//Example CSV output that can be extracted from the other information and plotted.
//unix/linux: grep "^CSV," > outfile.csv
ThermistorInside.ReadCalculate(3);
Serial.print("CSV,"); Serial.print(ThermistorInside.GetFarenheit(), 2);
Serial.print(","); Serial.print(ThermistorInside.GetCentigrade(), 2);
ThermistorOutside.ReadCalculate(3);
Serial.print(","); Serial.print(ThermistorOutside.GetFarenheit(), 2);
Serial.print(","); Serial.println(ThermistorOutside.GetCentigrade(), 2);

//Example of averaging 10 reads. Note that analogRead() isn't very fast.
//Part of this code has to be done manually, then call ReadADC() with the averaged number.
//Note that an unsigned int is only 16 bits. Do not exceed it or it will roll over and return trash.
ADCAverage = 0; //reset the variable for each loop.
for(i=0; i<10; i++) {
  ADCAverage += analogRead(THERMISTORPinInside);
  //delay(50); //add an extra delay to spread the average out a little more?
  }
ThermistorInside.ReadADC(ADCAverage/10); //call ReadADC() with the averaged value.
ThermistorInside.CalculateTemperature(3); //generate the temperature numbers.
//Report as normal from here on out.
Serial.print("ThermistorInside Averaged (Centigrade, Farenheit): ");
Serial.print(ThermistorInside.GetCentigrade(), 2);
Serial.print(", ");
Serial.println(ThermistorInside.GetFarenheit(), 2);

//Note: ReadCalculate() doesn't have to be called this much per loop, but it is shown for clarity.

//Timing how long some code runs...
//This could be done inside the class, but timings would be off.
Serial.println("Doing timing loop test...");
mytime1=millis(); //save the start time
for(i=0; i<1000; i++) {
   ThermistorInside.ReadCalculate(1);
   mytemp=ThermistorInside.GetFarenheit();
   }
mytime2 = millis() - mytime1;
Serial.print("1000 Farehneit reads in milliseconds (ReadCalculate(1)): ");
Serial.println(mytime2); //print out time

//Timing how long some code runs...
//This could be done inside the class, but timings would be off.
Serial.println("Doing timing loop test...");
mytime1=millis(); //save the start time
for(i=0; i<1000; i++) {
   ThermistorInside.ReadCalculate(2);
   mytemp=ThermistorInside.GetFarenheit();
   }
mytime2 = millis() - mytime1;
Serial.print("1000 Farehneit reads in milliseconds (ReadCalculate(2)): ");
Serial.println(mytime2); //print out time

//Timing how long some code runs...
//This could be done inside the class, but timings would be off.
Serial.println("Doing timing loop test...");
mytime1=millis(); //save the start time
for(i=0; i<1000; i++) {
   ThermistorInside.ReadCalculate(3);
   mytemp=ThermistorInside.GetFarenheit();
   }
mytime2 = millis() - mytime1;
Serial.print("1000 Farehneit reads in milliseconds (ReadCalculate(3)): ");
Serial.println(mytime2); //print out time

//Wait 10 seconds to avoid screen flooding.
delay(10000);

} //end loop()
```

```
//end of Thermistor4c.pde
```

Share |