Temperature Measurement with a Thermistor and an Arduino

Class Notes for EAS 199B

Gerald Recktenwald*

July 11, 2010

1 Measuring the Thermistor Signal

A thermistor indicates temperature by a change in electrical resistance. The analog input pins of the Arduino can only measure voltage, so the electrical resistance of a thermistor cannot be measured directly[1]. A simple technique for converting the (variable) resistance of the thermistor to a voltage is to use the thermistor in a voltage divider, as shown in the left side of Figure 1. The right side of Figure 1 is a representation of the voltage divider circuit connected to an Arduino.

The voltage divider has two resistors in series. In the left side of Figure 1, the upper resistor is the thermistor, with variable resistance R_t. The lower resistor has a fixed resistance R. A voltage V_s is applied to the top of the circuit[2]. The output voltage of the voltage divider is V_o, and it depends on V_s and R, which are known, and R_t, which is variable and unknown. As suggested by the wiring diagram in the right side of Figure 1, V_o is measured by one of the analog input pins on the Arduino.

1.1 The Voltage Divider

A voltage divider is a standard configuration of resistors. In this section we derive the simple equation that relates V_0, V_s, R and R_t.

The two resistors in series share the same current, I. The voltage drop across the two resistors is

$$V_s = I(R + R_t) \tag{1}$$

*Mechanical and Materials Engineering Department,Portland State University,Portland, OR, 97201, gerry@me.pdx.edu

[1]Electrical resistance is always measured indirectly, usually by inferring the resistance from a measured voltage drop. Precision resistance measurements involve techniques different from those described in these notes. Multimeters and other instruments that measure resistance have built-in circuits for measuring the voltage drop across a resistor when a precisely controlled and measured current is applied to the resistor.

[2]In the instructions for its data logging shield, adafruit recommends using the 3.3V input to power sensors because because the 5V line is noisey. To use the 3.3V line, the 3.3V signal is tied to the `Aref` pin. See `http://www.ladyada.net/make/logshield/lighttemp.html`. In my informal experiments, there was no apparent difference between the 3.3V and 5V supplies.

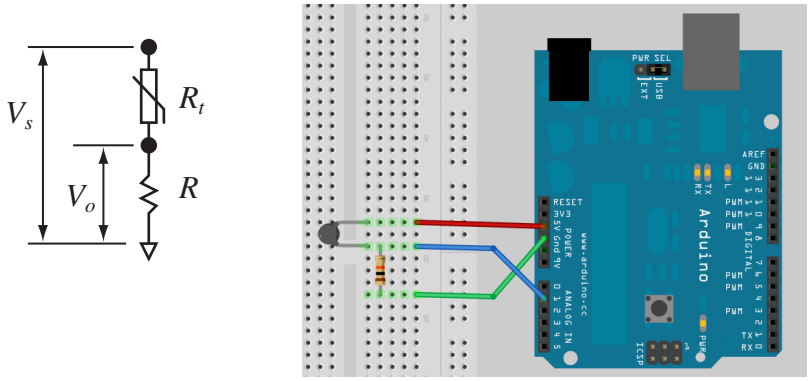

Figure 1: Voltage divider circuit (left) and sample breadboard wiring (right) for measuring voltage to indicate thermistor resistance. For the breadboard wiring on the right, V_o is measured with analog input Pin 1.

and the voltage drop across the fixed resistor is

$$V_o = IR \tag{2}$$

Solve Equation (1) and Equation (2) for I and set the resulting equations equal to each other

$$I = \frac{V_s}{R + R_t} \quad \text{and} \quad I = \frac{V_o}{R} \quad \Longrightarrow \quad \frac{V_s}{R + R_t} = \frac{V_o}{R}$$

The last equality can be rearranged to obtain.

$$\frac{V_o}{V_s} = \frac{R}{R + R_t} \tag{3}$$

Rearranging Equation (3) to solve for R_t gives

$$R_t = R\left(\frac{V_s}{V_o} - 1\right). \tag{4}$$

Equation (4) is used to compute the thermistor resistance from the measurement of V_o.

2 Measuring R_t and T with an Arduino

In this section a series of Arduino sketches are developed for measuring the thermistor output. The reader is assumed to be familiar with writing and running code on an Arduino. The sketches start with a simple printing of voltage and resistance, and progress to a reusable thermistor object that can easily be incorporated into other projects.

The sketches presented here can be adapted to work with any thermistor. The specific implementation here uses an Cantherm MF52A103J3470 NTC thermistor with a nominal resistance of 10 kΩ at 21 °C. The fixed resistor is a nominal 10 kΩ resistor. For any specific thermistor you will need a fixed resistor

of nominally equal resistance, and a calibration equation for resistance as a function of temperature.

The code for these sketches was developed after reading the sample code on the Arduino support site[3].

2.1 First Step: Measuring V_o and Computing R_t

The `ThermistorVoltageResistance.pde` sketch in Listing 2.1 measures and prints V_o, the voltage across the fixed resistor. The thermistor resistance computed from Equation (4) is also printed.

There are two main sections to `ThermistorVoltageResistance.pde`: the `setup` function and the `loop` function. These two functions are required in all Arduino sketches. The `setup` function establishes communication parameters with the host computer before running the main code.

```
void setup() {
  Serial.begin(9600);        // open serial port and set data rate to 9600 bps
  Serial.println("Thermistor voltage and resistance measurement:");
  Serial.println("\n Vs   Vo     Rt");
}
```

The `Serial.begin(9600)` statement sets the communication speed between the board and the host computer to be 9600 bits per second. The two `Serial.println` statements add labels to the serial monitor screen at the start of program execution.

The `loop` function is where the main work of the program occurs. The `loop` function begins with variable declarations and the assignment of constants.

```
int   ThermistorPin = 1;   //  Analog input pin for thermistor voltage
int   Vo;                  //  Integer value of voltage reading
float R = 9870.0;          //  Fixed resistance in the voltage divider
float Rt;                  //  Computed resistance of the thermistor
```

The voltage reading is stored in `Vo` as an integer because the `analogRead` function returns integers between 0 and 1023. The `R` and `Rt` variables are floating point type because we want store the result of computing R_t with Equation (4) using maximum precision.

The middle of the `loop()` function consists of the call to the `analogRead` function, and the direct evaluation of Equation (4).

```
Vo = analogRead(SensePin);
Rt = R*( 1023.0 / float(Vo) - 1.0 );
```

The evaluation of `Rt` is a direct translation of Equation (4). On a 10 bit scale, 1023.0 is the value corresponding to V_s, the maximum possible value of the voltage, which is 5V. The `float(Vo)` expression converts the integer value stored in `Vo` to floating point numbers before the division is performed. Consider the difference between integer and floating point math when `Vo` = 478.

$$\text{Integer math:} \quad \frac{1023}{467} - 1 \quad = \quad 2 - 1 = 1$$

$$\text{Floating point math:} \quad \frac{1023.0}{\texttt{float}(467)} - 1.0 \quad = \quad 2.1927 - 1.0 = 1.1927$$

[3]See `http://www.arduino.cc/playground/ComponentLib/`. In particular, the code here is most similar to the second version of the thermistor implementation, `http://www.arduino.cc/playground/ComponentLib/Thermistor2`.

```
//  File:  ThermistorVoltageResistance.pde
//
//  Use a voltage divider to indicate electrical resistance of a thermistor.

// -- setup() is executed once when sketch is downloaded or Arduino is reset
void setup() {
  Serial.begin(9600);     // open serial port and set data rate to 9600 bps
  Serial.println("Thermistor voltage and resistance measurement:");
  Serial.println("\n   Vo     Rt");
}

// -- loop() is repeated indefinitely
void loop() {
  int   ThermistorPin = 1;   //  Analog input pin for thermistor voltage
  int   Vo;                  //  Integer value of voltage reading
  float R = 9870.0;          //  Fixed resistance in the voltage divider
  float Rt;                  //  Computed resistance of the thermistor

  Vo = analogRead(ThermistorPin);
  Rt = R*( 1023.0 / float(Vo) - 1.0 );

  Serial.print("  ");  Serial.print(Vo);
  Serial.print("  ");  Serial.println(Rt);
  delay(200);
}
```

Listing 1: ThermistorVoltageResistance.pde

Performing the division as floating point numbers instead of integers prevents rounding and therefore preserves precision.

Running `ThermistorVoltageResistance.pde` produces the following output in the Serial Monitor[4].

```
Thermistor voltage and resistance measurement:

   Vo     Rt
  487  10863.08
  488  10820.59
  488  10820.59
  487  10863.08
  488  10820.59
  489  10778.28
...
```

Notice how the resolution of the analog to digital converter (ADC) on the Arduino causes discontinuous jumps in the measured value of R_t. When the value of `Vo` changes from 487 to 488, the value of `Rt` jumps from $10863\,\Omega$ to $10820\,\Omega$. The jump is due to a change in the least significant bit of the value obtained by the 10 bit ADC, not because of a discontinuous variation in the physical resistance. Conversion of the measured `Rt` values to temperature will result in a corresponding jump in T values. Appendix A contains an analysis of the temperature resolution limit of the Arduino as a function of R_t and R.

[4]The specific values of `Vo` and `Rt` will vary from run to run, and will depend on the characteristics of the fixed resistor and thermistor used, and on the temperature of the thermistor.

```
//  File:  ThermistorTemperature.pde
//
//  Use a voltage divider to indicate electrical resistance of a thermistor.
//  Convert the resistance to temperature.

// -- setup() is executed once when sketch is downloaded or Arduino is reset
void setup() {
  Serial.begin(9600);        // open serial port and set data rate to 9600 bps
  Serial.println("Thermistor temperature measurement:");
  Serial.println("\n   Vo     Rt     T (C)");
}

// -- loop() is repeated indefinitely
void loop() {
  int   ThermistorPin = 1;   //  Analog input pin for thermistor voltage
  int   Vo;                  //  Integer value of voltage reading
  float R = 9870.0;          //  Fixed resistance in the voltage divider
  float logRt,Rt,T;
  float c1 =  1.009249522e-03, c2 = 2.378405444e-04, c3 = 2.019202697e-07;

  Vo = analogRead(ThermistorPin);
  Rt = R*( 1023.0 / (float)Vo - 1.0 );
  logRt = log(Rt);
  T = ( 1.0 / (c1 + c2*logRt + c3*logRt*logRt*logRt ) ) - 273.15;

  Serial.print("  ");  Serial.print(Vo);
  Serial.print("  ");  Serial.print(Rt);
  Serial.print("  ");  Serial.println(T);
  delay(200);
}
```

Listing 2: ThermistorTemperature.pde

2.2 Computing T from R_t

As described in the thermistor calibration exercise, the thermistor temperature can be computed with the Steinhart-Hart equation

$$T = \frac{1}{c_1 + c_2 \ln(R) + c_3 (\ln(R))^3} \tag{5}$$

For the thermistor used in this demonstration, $c_1 = 1.009249522 \times 10^{-3}$, $c_2 = 2.378405444 \times 10^{-4}$, $c_3 = 2.019202697 \times 10^{-7}$.

The `ThermistorTemperature.pde` sketch in Listing 2 incorporates the evaluate of Equation (5) into the Arduino code. The substantial changes from `ThermistorVoltageResistance.pde` are the definition of the constants for the Steinhart-Hart equation in the `local` function,

```
float c1 =  1.009249522e-03, c2 = 2.378405444e-04, c3 = 2.019202697e-07;
```

and the evaluation of T with the Steinhart-Hart equation

```
logRt = log(Rt);
T = ( 1.0 / (c1 + c2*logRt + c3*logRt*logRt*logRt ) ) - 273.15;
```

Running `ThermistorTemperature.pde` produces the following output in the Serial Monitor.

```
Thermistor temperature measurement:
```

```
  Vo     Rt     T (C)
 486  10905.74  22.47
 482  11078.15  22.07
 485  10948.58  22.37
 484  10991.59  22.27
 485  10948.58  22.37
 485  10948.58  22.37
 485  10948.58  22.37
 483  11034.78  22.17
...
```

This output shows that a change of one unit in `Vo` causes a change of 0.1 °C in the computed temperature. Thus, we can infer that the temperature resolution of this thermistor/aduino combination is 0.1 °C. Note that resolution is only a measure of the smallest difference that a sensor and instrument combination can detect. A resolution of 0.1 °C does *not* imply that the *accuracy* of the measurement is ±0.1 °C. The temperature resolution is discussed further in Appendix A.

3 Modular Code for Thermistor Measurements

The code in the sketches in Listing 2.1 and Listing 2 are practically useful. For measuring the output of a single thermistor, there is no intrinsic need to modify the code further, except as may be necessary to adjust the c_i coefficients and the value of fixed resistor R.

As the complexity of Arduino programs increases, there is a benefit to introducing more structure to the software. In this section and the next, the thermistor measuring code will be isolated into modules that can easily be added to new projects. The short term cost is that the code becomes more a little more complicated. The long term benefit is that once the code is developed, it can be reused in other projects without having to dig into the details of the voltage-to-temperature conversion. The sidebar to the right uses an analogy to explain the benefits of creating modular and reusable code.

Reusable code and task delegation

The use of modular code for temperature measurement can be imagined as a conversation between two workers, a lab manager a temperature measurement expert, T Expert.

Lab Manager:	What is the temperature of the sensor on pin 1?
T Expert:	22.3 °C

In this imaginary conversation, Lab Manager is responsible for performing a number of tasks (reading values, making decisions, adjusting settings) and does not want to get bogged down in the details of temperature measurement. T Expert is focused on one job: reading the temperature of a thermistor sensor. T Expert can (and does) work with other lab managers on other projects.

Because T Expert has refined his technique for temperature measurement, Lab Manager has come to trust the values returned by T Expert. T Expert can improve his measurement process, for example by using a more accurate calibration equation, but any improvements to the measurement process do not (need to) change the transaction: Lab Manager still askes for the temperature on pin 1, and T Expert still returns a value, the temperature.

At one time, Lab Manager was responsible for making temperature measurements by herself, but as her responsibilities grew and became more complex, she found that she could do her job more efficiently and with fewer errors if she delegated the details of the temperature measurement task to T Expert.

Figure 2 is a common example of a reusable code module, namely the $\log(x)$ function from the standard mathematics library. Do you know how the arduino `log(x)` function works? Do you know how many internal variables (like the c_i in the thermistor calibration)

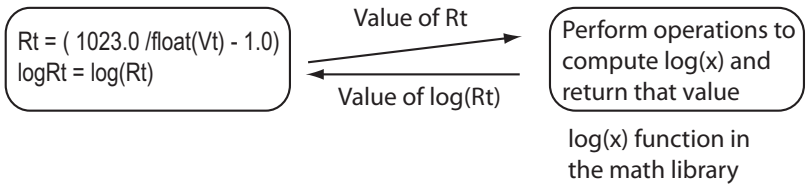

Figure 2: The mathematics library has several reusable functions.

and what kinds of operations are required to obtain a value for `log(x)`? The answer to these questions is, "Of course not." Or perhaps you would think, "Why should I care as long as the correct result is returned?". Those answers make sense and reflect the decisions to delegate tasks to people or services or utilities that we trust.

With the example of `log(x)` in mind, what are the desirable characteristics of a reusable module that allows us to measure the output of a thermistor? There are many valid answers. In the next section we will develop a simple model of a reusable thermistor module where all the code resides in a single function. In another section, a more flexible, *object oriented* module will be developed.

We really have two goals. First, we want to create a reusable module for thermistor measurements. Reusable code is valuable for current and future applications of temperature measurement with an Arduino. Second, we want to understand some of the general principles for reusable code development. The use of modular, reusable code is good practice in any programming work.

3.1 A Reusable Thermistor Function

In this section a function (or subroutine) for measuring thermistor temperature is developed. The function has one input, the integers identifying the pin measuring the output for the thermistor voltage divider. The function has a single return value, the temperature of the thermistor.

The code for the thermistor function is in a separate file, `thermistorFun.cpp` in Listing 3. To use this function in a sketch, you need to refer to its code interface – it's definition of input and output parameters – in the sketch *before* the function is called. The code interface (also called a *function prototype*) is in yet another file `thermistorFun.h`, which his only one line long and which is shown in Listing 4. The `thermistorFun.h` and `thermistorFun.cpp` files constitute a reusable code module for performing thermistor temperature readings with an Arduino. The code in `thermistorFun.cpp` embodies the details of the temperature measurement from the `thermistorTemperature.pde` sketch.

The `ThermistorTemperatureFunction.pde` sketch has two references to the `thermistorFun` function. At the start of the sketch is the line

```
#include "thermistorFun.h"
```

The `#include` directive tells the compiler to read the contents of the file `thermistorFun.h` *as if* that file were part of the `thermistorFun.pde`.

The temperature measurement is performed by the single line

```
T = thermistorRead(ThermistorPin);
```

```
//  File:  thermistorFun.cpp
//
//  Read a thermistor with Arduino and return temperature

#include "WProgram.h"
#include "thermistorFun.h"

float thermistorRead(int Tpin) {
  int   Vo;
  float logRt,Rt,T;
  float R = 9870;          //  fixed resistance, measured with multimeter
  //  c1, c2, c3 are calibration coefficients for a particular thermistor
  float c1 =  1.009249522e-03, c2 = 2.378405444e-04, c3 = 2.019202697e-07;

  Vo = analogRead(Tpin);
  Rt = R*( 1024.0 / float(Vo) - 1.0 );
  logRt = log(Rt);
  T = ( 1.0 / (c1 + c2*logRt + c3*logRt*logRt*logRt ) ) - 273.15;

  return T;
}
```

Listing 3: The `ThermistorFun.cpp` code contains the C code function for thermistor measurements.

```
//  File:  thermistorFun.h
//  Header file defining prototype for functions that read thermistors

float thermistorRead(int Tpin);
```

Listing 4: The `ThermistorFun.h` header file defines C function prototype for thermistor measurements.

which stores the thermistor temperature in the variable `T`. The `thermistorRead` function hides the details of performing the temperature measurement. This makes the code in the sketch easier to read and understand.

Running `ThermistorTemperatureFunction.pde` produces the following output in the Serial Monitor.

```
Thermistor temperature measurement:

T (C)
22.52
22.72
22.62
22.72
22.62
22.52
22.62
...
```

```
//  File:  ThermistorTemperatureFunction.pde
//
//  Use a voltage divider to indicate electrical resistance of a thermistor.
//  Thermistor reading and conversion calculations are encapsulated in a
//  reusable function.

#include <thermistorFun.h>

// -- setup() is executed once when sketch is downloaded or Arduino is reset
void setup() {
  Serial.begin(9600);        // open serial port and set data rate to 9600 bps
  Serial.println("Thermistor temperature measurement:");
  Serial.println("\nT (C)");
}

// -- loop() is repeated indefinitely
void loop() {
  int   ThermistorPin = 1;     //  Analog input pin for thermistor voltage
  float T;                     //  Value of temperature reading

  T = thermistorRead(ThermistorPin);
  Serial.println(T);
  delay(200);
}
```

Listing 5: ThermistorTemperatureFunction.pde

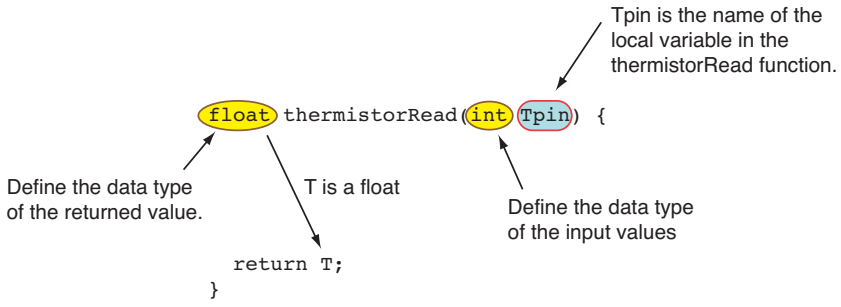

Figure 3: Defining the variable types for input and output of the `ThermistorFun` function.

3.2 A Reusable Thermistor Object

In this section we develop a $C++$ object for thermistor measurements with an Arduino. This object-oriented approach improves on the procedural function introduced in the preceding section by easily allowing for multiple thermistors to be used in the same sketch. It also allows the calibration coefficients and the (measured) value of the fixed resistance to be specified for each of the thermistors. Similar functionality could be added to the procedural (C code) implementation, but not as elegantly. For a terse introduction to using $C++$ classes in Arduino development, See `http://arduino.cc/en/Hacking/LibraryTutorial`.

Table 1 lists the source files that constitute the reusable thermistor objects. Table 2 lists the public methods that the object oriented module provides.

Table 1: Files in the $C++$ implementation of thermistor measurement.

`thermistor.cpp`	Class implementation
`thermistor.h`	Class specification/definition
`thermistorSensor.pde`	Sketch demonstrating of using the object-oriented interface.

Table 2: Objects in the $C++$ implementation of thermistor measurement.

`Thermistor`	Constructor. Use this to define a thermistor object. Example: `th = Thermistor(1)` creates a thermistor object `th` that reads the voltage divider output on pin 1.
`fixedResistance`	Public method to change the value of the fixed resistance in a thermistor object. Example: `th.fixedResistance(9942)` changes the fixed resistance for the `th` object to $9942\,\Omega$.
`coefficients`	Public method to change a the coefficients in the Steinhart-Hart calibration equation for a thermistor object Example: `th.coefficients(1.23e-4,5.67e-5,8.90e-6)` changes the calibration coefficients of the `th` object to $c_1 = 1.23 \times 10^{-4}$, $c_2 = 5.67 \times 10^5$, and $c_3 = 8.90 \times 10^6$.
`getTemp`	Public method to read the temperature of a thermistor object. Example: `T = th.getTemp()` reads the thermistor connected to the `th` object.

Running `thermistorSensor` produces the following output

```
T (C)
22.77
22.57
```

Table 3: Code sizes for different implementations of the Arduino temperature measurement.

Code	Size (Bytes)
`thermistorVoltageResistance.pde`	4004
`thermistorTemperature.pde`	4504
`thermistorTemperatureFunction.pde`	4350
`thermistor.pde`	4514

```
22.77
22.67
22.57
22.57
22.67
22.67
22.57
```

Not Done Yet: Continue Here

```
//  File: Thermistor.cpp
//
//  Use a voltage divider to indicate electrical resistance of a thermistor
//  Thermistor reading and T conversion are encapsulated in reusable objects
//
//  Based on code by Max Mayfield, max.mayfield@hotmail.com
//  posted on the Arduino playground
//  http://www.arduino.cc/playground/ComponentLib/Thermistor2
//  Modified and extended by G. Recktenwald, recktenwaldg@gmail.com
//  2010-06-07

#include "WProgram.h"
#include "Thermistor.h"

// --- constructor
Thermistor::Thermistor(int pin) {
  _pin = pin;              //  Analog input pin to read voltage divider output
  _Rfix = 9970;            //  Resistance of fixed resistor in the divider
  _a1 = 1.009249522e-03;   //  Coefficients in the Steinhart-Hart equation
  _a2 = 2.378405444e-04;
  _a3 = 2.019202697e-07;
}

// --- Set the value of the fixed resistance
void Thermistor::fixedResistance(float Rfix) {
  _Rfix = Rfix;
}

// --- Set values of coefficients in the Steinhart-Hart equation
void Thermistor::coefficients(float a1, float a2, float a3) {
  _a1 = a1;
  _a2 = a2;
  _a3 = a3;
}

// --- Read the voltage across the fixed resistance, and from it compute T
float Thermistor::getTemp() {

  int vin  = analogRead(_pin);     //  voltage corresponding to thermistor Rt
  float Rt;
  float logRt, T;

  Rt = _Rfix * ( ( 1023.0 / float(vin) ) - 1.0 );
  logRt = log(Rt);
  T = ( 1.0 / (_a1 + _a2*logRt + _a3*logRt*logRt*logRt ) ) - 273.15;

  return T;
}
```

Listing 6: The `Thermistor.cpp` code contains the class implementation for objected-oriented thermistor measurements.

```
//  File: Thermistor.h
//
//  Use a voltage divider to indicate electrical resistance of a thermistor
//  Thermistor reading and T conversion are encapsulated in reusable objects
//
//     Vdd ---o    Vdd is supply voltage, typically 5V
//            |
//            Rt   (thermistor)
//            |
//      Vo ---o    Vo is output from the voltage divider:  Vo = Vdd*R/(R+Rt)
//            |
//            R    (fixed resistor, R approx. equal to Rt
//            |
//           GND
//
//  Based on code by Max Mayfield, max.mayfield@hotmail.com
//  posted on the Arduino playground
//  http://www.arduino.cc/playground/ComponentLib/Thermistor2
//  Modified and extended by G. Recktenwald, recktenwaldg@gmail.com
//  2010-06-07

#ifndef Thermistor_h

#define Thermistor_h
#include "WProgram.h"
#include "math.h"

class Thermistor {

  public:
    Thermistor(int pin);
    float  getTemp();
    void   fixedResistance(float Rfix);
    void   coefficients(float a1, float a2, float a3);

  private:
    int    _pin;
    float  _Rfix;
    float  _a1, _a2, _a3;
};

#endif
```

Listing 7: The `Thermistor.h` header file defines the class interface for object-oriented thermistor measurements.

```
//  File:  ThermistorSensor.pde
//
//  Use a voltage divider to indicate electrical resistance of a thermometer
//  Measurements and conversions are managed via objects

#include <Thermistor.h>

// --- temp is a thermistor object
Thermistor temp(1);

// -- setup() is executed once when sketch is downloaded or Arduino is reset
void setup() {
  Serial.begin(9600);
  temp.fixedResistance(9870);
  temp.coefficients( 1.009249522e-03, 2.378405444e-04, 2.019202697e-07);
  Serial.println("T (C)");
}

// -- loop() is repeated indefinitely
void loop() {
  double temperature = temp.getTemp();

  Serial.println(temperature);
  delay(200);
}
```

Listing 8: ThermistorSensor.pde

A Temperature resolution

In this Appendix we consider how the resolution of the Analog to Digital Converter and the choice of the fixed resistor can affect the temperature resolution of the circuit in Figure 1.

Thermistors can be characterized by their resistance at 21 °C. We will designate that resistance R_T^*

$$R_T^* = R(T = 21\,^\circ\mathrm{C}) \tag{6}$$

The Analog to Digital Converter (ADC) on the Arduino board has 10 bits of resolution: the range of voltage inputs is divided into $2^{10} = 1024$ levels. The standard input range is $V_s = 5\,\mathrm{V}$ so the standard resolution of the input ADC is

$$\delta V = \frac{V_s}{1024} = 4.88\,\mathrm{mV} \tag{7}$$

How does this voltage resolution determine the temperature resolution of the Arduino when measuring thermistor output with the circuit in Figure 1? If the voltage reading has a resolution of $\pm\delta V$, then the uncertainty in voltage is $\pm\delta V/2$. A change in voltage corresponds to a change in resistance.

Since

$$R_T^* = R_T \left(\frac{V_s}{V_o^*} - 1 \right)$$

then

$$R_T^* + \delta R = R \left(\frac{V_s}{V_o^* + \delta V} - 1 \right) \tag{8}$$

$$T^* + \delta T = f(R_T^* + \delta R) \tag{9}$$

Therefore

$$\delta T = (T^* + \delta T) - T^* = f(R_T^* + \delta R) - f(R_T^*) \tag{10}$$

For the thermistor in this demonstration, $dT(T = T^*) = 0.098\,^\circ\mathrm{C}$

Figure 4 is a plot of δT as a function of fixed resistance R. The minimum resolution (which is desired) is obtained when $R = R_T^*$. However, the shape of the $\delta T = f(R)$ function is rather flat near R_T^*. Therefore, the voltage divider is relatively forgiving of mismatch between R and R_T^*.

It must be stressed that δT is *not* the accuracty of the temperature reading. δT is only the resolution (or precision) of the temperature reading.

B Issues to Consider

B.1 ATMEL chips want low impedance on ADC inputs

From this thread on the Adafruit forum

`http://forums.adafruit.com/viewtopic.php?f=25&t=15744`

`fat16lib` quotes the ATmega328 data sheet, section 23.6.1:

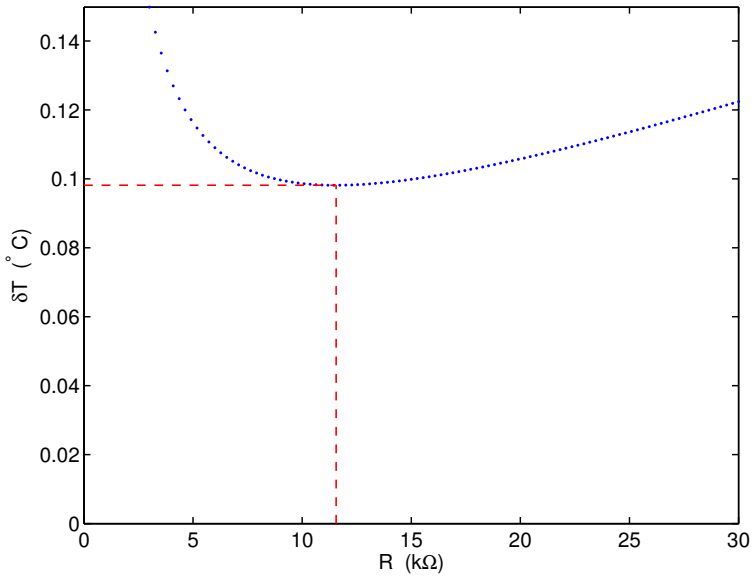

Figure 4: Variation in temperature resolution δT as a function of the fixed resistance R in the voltage divider for a given nominal R_T^*.

> The ADC is optimized for analog signals with an output impedance of approximately 10 kΩ or less.
>
> The user is recommended to only use low impedance sources with slowly varying signals, since this minimizes the required charge transfer to the S/H capacitor.

`fat16lib` then refers to another thread on the Adafruit forum where an 18-bit external ADC is discussed

> `http://forums.adafruit.com/viewtopic.php?f=31&t=12269`

B.2 Using an external, high precision ADC

`http://forums.adafruit.com/viewtopic.php?f=25&t=15744#p77858`

B.3 Using an op-amp voltage buffer to reduce input impedance

Wikipedia article giving an overview of the buffer amplifier

> `http://en.wikipedia.org/wiki/Buffer_amplifier`

National Instruments has a tutorial on selecting the components for a voltage-following buffer

> `http://zone.ni.com/devzone/cda/tut/p/id/4494`