Temperature Measurement with a Thermistor and an Arduino

Class Notes for EAS 199B

Gerald Recktenwald∗

May 25, 2013

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. 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. 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 $V_o$, $V_s$, $R$ and $R_t$.

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

$$V_s = I(R + R_t) \quad (1)$$
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

$$V_o = IR$$  \hspace{1cm} (2)

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

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

The last equality can be rearranged to obtain.

$$\frac{V_o}{V_s} = \frac{R}{R + R_t} \hspace{1cm} (3)$$

Rearranging Equation (3) to solve for $R_t$ gives

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

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

G. Recktenwald, gerry@me.pdx.edu

May 25, 2013
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\(^3\). The purpose of this document is to provide an introduction to Arduino programming by using the thermistor measurement as a case study. The pace will probably be too slow for readers already familiar with Arduino programming. Readers looking for code to incorporate into their sketches should focus on Section 3.1, A Reusable Thermistor Function and Section 3.2, A Reusable Thermistor Object.

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.

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

```cpp
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 \( R_t \) 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).

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

The evaluation of \( R_t \) 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 \( Vo = 478 \).

\(^3\)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

\[
\text{Integer math: } \frac{1023}{467} - 1 = 2 - 1 = 1
\]

\[
\text{Floating point math: } \frac{1023.0}{\text{float}(467)} - 1.0 = 2.1927 - 1.0 = 1.1927
\]

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:**

<table>
<thead>
<tr>
<th>Vo</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>487</td>
<td>10863.08</td>
</tr>
<tr>
<td>488</td>
<td>10820.59</td>
</tr>
<tr>
<td>488</td>
<td>10820.59</td>
</tr>
<tr>
<td>487</td>
<td>10863.08</td>
</tr>
<tr>
<td>488</td>
<td>10820.59</td>
</tr>
<tr>
<td>489</td>
<td>10778.28</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

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 \(V_o\) changes from 487 to 488, the value of \(R_t\) jumps from 10863Ω to 10820Ω. 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 characteristics of the fixed resistor and thermistor used, and on the temperature of the thermistor.

---

\(^4\)The specific values of \(V_o\) and \(R_t\) 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.
Use a voltage divider to indicate electrical resistance of a thermistor.

---

```
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)");
}

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

resistance. Conversion of the measured $R_t$ 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 $R$.

### 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

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

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 evaluation 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,

$$\text{float } c1 = 1.009249522e-03, c2 = 2.378405444e-04, c3 = 2.019202697e-07;$$

and the evaluation of $T$ with the Steinhart-Hart equation

$$\text{logRt = log(Rt);}
\text{T = ( 1.0 / (c1 + c2*logRt + c3*logRt*logRt*logRt ) ) - 273.15; }$$
Running *ThermistorTemperature.pde* produces the following output in the Serial Monitor.

**Thermistor temperature measurement:**

<table>
<thead>
<tr>
<th>V₀</th>
<th>Rₜ</th>
<th>T (°C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>486</td>
<td>10905.74</td>
<td>22.47</td>
</tr>
<tr>
<td>482</td>
<td>10788.15</td>
<td>22.07</td>
</tr>
<tr>
<td>485</td>
<td>10948.58</td>
<td>22.37</td>
</tr>
<tr>
<td>489</td>
<td>10991.59</td>
<td>22.27</td>
</tr>
<tr>
<td>485</td>
<td>10948.58</td>
<td>22.37</td>
</tr>
<tr>
<td>485</td>
<td>10948.58</td>
<td>22.37</td>
</tr>
<tr>
<td>485</td>
<td>10948.58</td>
<td>22.37</td>
</tr>
<tr>
<td>483</td>
<td>11034.78</td>
<td>22.17</td>
</tr>
</tbody>
</table>

This output shows that a change of one unit in V₀ causes a change of 0.1 °C in the computed temperature. Thus, we can infer that the temperature resolution of this thermistor/arduino 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 cᵢ 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.

Figure 2 is a common example of a reusable code module, namely

---

**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.

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**.

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 asks for the temperature on pin 1, and **T EXPERT** still returns a value, the temperature.
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) 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 is the 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

```pde
#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`.  

---

G. Recktenwald, gerry@me.pdx.edu  
May 25, 2013
#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.

The temperature measurement is performed by the single line

T = thermistorRead(TermistorPin);

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
...
Use a voltage divider to indicate electrical resistance of a thermistor. Thermistor reading and conversion calculations are encapsulated in a reusable function.

```cpp
#include <thermistorFun.h>

void setup() {
  Serial.begin(9600); // open serial port and set data rate to 9600 bps
  Serial.println("Thermistor temperature measurement:");
  Serial.println("\nT (C)");
}

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

![Diagram of thermistorRead function](image)

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

G. Recktenwald, gerry@me.pdx.edu

May 25, 2013
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.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thermistor.cpp</td>
<td>Class implementation</td>
</tr>
<tr>
<td>thermistor.h</td>
<td>Class specification/definition</td>
</tr>
<tr>
<td>thermistorSensor.pde</td>
<td>Sketch demonstrating of using the object-oriented interface.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thermistor</td>
<td>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.</td>
</tr>
<tr>
<td>fixedResistance</td>
<td>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 9942Ω.</td>
</tr>
<tr>
<td>coefficients</td>
<td>Public method to change 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 ).</td>
</tr>
<tr>
<td>getTemp</td>
<td>Public method to read the temperature of a thermistor object. Example: T = th.getTemp() reads the thermistor connected to the th object.</td>
</tr>
</tbody>
</table>

Running thermistorSensor produces the following output

<table>
<thead>
<tr>
<th>T (°C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.77</td>
</tr>
<tr>
<td>22.57</td>
</tr>
</tbody>
</table>

G. Recktenwald, gerry@me.pdx.edu

May 25, 2013
Table 3: Code sizes for different implementations of the Arduino temperature measurement.

<table>
<thead>
<tr>
<th>Code</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>thermistorVoltageResistance.pde</td>
<td>4004</td>
</tr>
<tr>
<td>thermistorTemperature.pde</td>
<td>4504</td>
</tr>
<tr>
<td>thermistorTemperatureFunction.pde</td>
<td>4350</td>
</tr>
<tr>
<td>thermistor.pde</td>
<td>4514</td>
</tr>
</tbody>
</table>

Not Done Yet: Continue Here
Listing 6: The Thermistor.cpp code contains the class implementation for object-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

We now 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.

The thermistors used in the fish tank project can be characterized by their resistance at 21 °C. We will designate that resistance \( R^*_T \):

\[
R^*_T = R(T = 21 \, ^\circ C) \quad (6)
\]

The Analog to Digital Converter (ADC) on the Arduino board has 10 bits of resolution: the range of voltage inputs is divided into \( 2^{10} - 1 = 1023 \) intervals. The standard input range is \( V_s = 5 \, V \) so the standard resolution of the input ADC is\(^5\)

\[
\delta V = \frac{V_s}{1023} = 4.89 \, mV \quad (7)
\]

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 \( \pm \delta V/2 \).

When the resistance of the thermistor changes, the voltage output of the voltage divider changes. From Equation (4),

\[
R^*_T = R_T \left( \frac{V_s}{V_o^*} - 1 \right)
\]

then

\[
R^*_T + \delta R = R \left( \frac{V_s}{V_o^* + \delta V} - 1 \right) \quad (8)
\]

\[
T^* + \delta T = f(R^*_T + \delta R) \quad (9)
\]

Therefore

\[
\delta T = (T^* + \delta T) - T^* = f(R^*_T + \delta R) - f(R^*_T) \quad (10)
\]

For the thermistor in this demonstration, \( dT(T = T^*) = 0.099 \, ^\circ C \). Therefore, we say that the precision of the 10-bit reading at the nominal resistance is approximately 0.1 °C. In other words, due to the resolution of the of ADC, which from Equation (7) is 4.9 mV, the smallest temperature change that can be detected is 0.1 °C.

Figure 4 is a plot of the precision, \( \delta T \), as a function of the 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 \( \delta T \) is not the accuracy of the temperature reading, \( \delta T \) is only the resolution (or precision) of the temperature reading.

---

\(^5\)The maximum 10-bit number is \( 2^{10} - 1 = 1023 \). If the number line is divided into segments labeled 0, 1, 2, . . . , 1023, there are 1024 values (1024 tick marks) and 1023 intervals. Therefore, the correct scaling is \( 5/1023 \), not \( 5/1024 \).
Figure 4: Variation in temperature resolution $\delta T$ as a function of the fixed resistance $R$ in the voltage divider for a given nominal $R^*$.  

B  Issues to Consider

B.1  ATMEL chips want low impedance on ADC inputs

From this thread on the Adafruit forum


fat16lib quotes the ATmega328 data sheet, section 23.6.1:

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


B.2 Using an external, high precision ADC


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

Wikipedia article giving an overview of the 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