ANALOG DEVICES

AN-709
APPLICATION NOTE

One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106 • Tel: 781/329-4700 • Fax: 781/326-8703 • www.analog.com

RTD Interfacing and Linearization Using an ADuC8xx MicroConverter®

by Grayson King and Toru Fukushima

INTRODUCTION

The platinum RTD is one of the most accurate sensors available for measuring temperature within the range –200°C to +850°C, capable of achieving calibrated accuracy of ±0.02°C or better. To get the most of an RTD's accuracy, however, requires precise signal conditioning, A/D conversion, linearization, and calibration. The Analog Devices MicroConverter product family (http://www.analog.com/MicroConverter) includes devices with a 16-bit or 24-bit ADC and an 8052 MCU in a single chip along with signal conditioning circuitry ideally suited to RTD sensors. This application note describes ways to implement a complete RTD sensor interface using the ADuC834 (or other similar MicroConverter) and just a few passive components.

Software utilities and sample code, referenced here and highly recommended for anyone implementing a MicroConverter-based RTD sensor interface, can be found at http://www.analog.com/MicroConverter

HARDWARE DESIGN

An RTD (resistance temperature detector) is a resistance that varies as a function of temperature in a precisely defined manner. Before getting into the details of the RTD's transfer function of resistance to temperature (which is nonlinear), assume that the nonlinearities will be corrected digitally, and first concentrate on converting the RTD's resistance to a digital value. A common way to do this is shown in Figure 1.

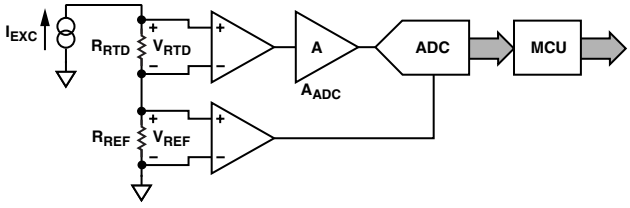

Figure 1. RTD Interfacing Hardware Configuration

Here, a single current source (I_{EXC}) excites both the RTD (R_{RTD}) and a precision reference resistor (R_{REF}) by way of a series connection, generating the ADC input voltage (V_{RTD}) and reference voltage (V_{REF}), respectively:

$$V_{RTD} = I_{EXC} \times R_{RTD}$$
$$V_{REF} = I_{EXC} \times R_{REF}$$

The ADC's normalized digital output (zero input = 0, full-scale input = 1) is simply a ratio of its input voltage to its reference voltage multiplied by the gain stage, A_{ADC}:

$$ADC_{norm} = A_{ADC} \times \frac{V_{RTD}}{V_{REF}} = A_{ADC} \times \frac{I_{EXC} \times R_{RTD}}{I_{EXC} \times R_{REF}} = A_{ADC} \times \frac{R_{RTD}}{R_{REF}}$$

Notice how I_{EXC} cancels out of the above equation, meaning that even if the excitation current changes or is imprecise, the ADC result always corresponds directly to the ratio of the RTD resistance to the reference resistance. Choosing a precision, low-drift reference resistor means the RTD resistance can be known to a high degree of precision, even with a much less precise current source.

Applying this same principle using a MicroConverter, Figure 2 shows the ADuC834 connected for interfacing with a 4-wire RTD. Note that this is the same overall topology as shown in Figure 1, except that all the active components (excitation current source, differential input stages for V_{RTD} and V_{REF}, gain stage A_{ADC}, the ADC itself, and a microcontroller) are included internally to the ADuC834 chip, along with a number of other peripherals such as serial communication ports for the digital communication path(s). Notice also that some passive components have been added for R/C filtering of signals and for protection from overvoltage conditions at the terminal block. This represents a complete implementation, requiring only a power supply and any particular peripheral chip needed for the digital interface (RS-232 or RS-485 line driver/receiver, for example).

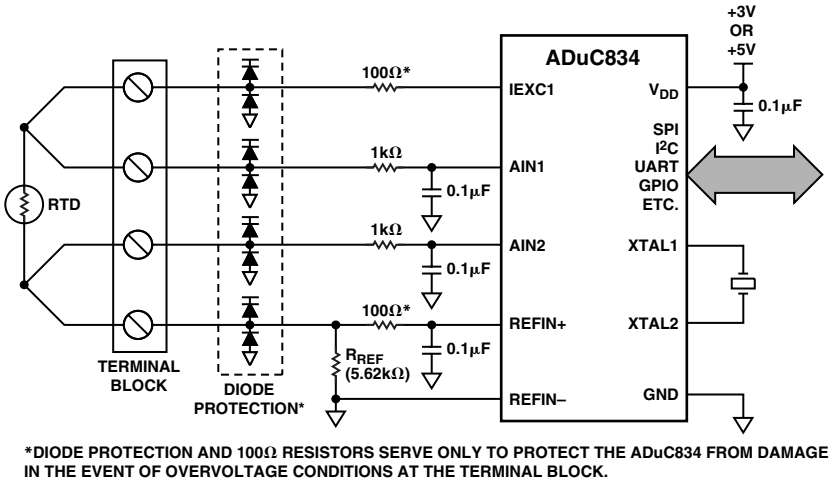

Figure 2. Complete RTD Interfacing Circuit Using ADuC834

CALCULATING RTD RESISTANCE FROM ADC RESULT

Recall from the Hardware Design section that

$$ADC_{norm} = A_{ADC} \times \frac{R_{RTD}}{R_{REF}}$$

which can be rewritten as

$$R_{RTD} = ADC_{norm} \times \frac{R_{REF}}{A_{ADC}} = ADC_{norm} \times scale$$

where:

$$scale = \frac{R_{REF}}{A_{ADC}}$$

The *scale* value is the fixed scaling factor used in the sample code. Taking this a step further, a fixed offset value can be added to the equation, resulting in

$$R_{RTD} = ADC_{norm} \times scale + offset$$

where the *offset* term represents a fixed offset that can be used to compensate for errors. This *offset* term is discussed further in the Calibration section. For most situations, a value of zero is sufficient for this *offset* term. Note that a direct equation for RTD resistance has been obtained as a function of the ADC result using only a pair of fixed values for *scale* and *offset*.

The remainder of this document considers the most common type of platinum RTD, which has a nominal resistance (R_0) of 100 Ω at 0°C. Also assume a reference resistor value of 5.62 kΩ, which provides a good match to such an RTD. With these component values and using the ADuC834, an internal gain of 7.8125 is the highest available ADC gain setting that still allows the RTD to cover its full specified temperature range. (Recall that ADC_{norm} is by definition limited to the range 0 to 1, which is what defines the temperature range limitation at higher ADC gains.) The gain of 7.8125 corresponds to an ADC0CON value of 0x4C, or a range setting of 320 mV unipolar ($A_{ADC} = V_{REF}$/span = 2.5 V/320 mV = 7.8125). To correspond to this gain setting, the scale value works out to 719.36 (scale = R_{REF}/A_{ADC} = 5.62 k/7.8125 = 719.36,) which is the default scale value used in the sample code. The default value for the offset term is zero.

The above equations for R_{RTD} are merely methods of determining (in software) the RTD's resistance directly from a given ADC conversion result. To then determine the RTD's temperature as a function of its resistance requires an understanding of the RTD's transfer function.

RTD TRANSFER FUNCTION

A platinum RTD's transfer function is described by two distinct polynomial equations: one for temperatures below 0°C and another for temperatures above 0°C. These equations are

$$R_{RTD}(t) = R_0[1 + At + Bt^2 + C(t - 100°C)t^3] \quad \text{(for } t \leq 0°C\text{)}$$

$$R_{RTD}(t) = R_0[1 + At + Bt^2] \quad \text{(for } t \geq 0°C\text{)}$$

where:

t = RTD temperature [°C]

$R_{RTD}(t)$ = RTD resistance as a function of RTD temperature *(t)*

R_0 = RTD resistance at 0°C (most often 100 Ω)

$A = 3.9083 \times 10^{-3}$ °C^{-1}

$B = -5.775 \times 10^{-7}$ °C^{-2}

$C = -4.183 \times 10^{-12}$ °C^{-4}

Notice that the notation is changed from R_{RTD} to $R_{RTD}(t)$

to reflect that the RTD's resistance is a function of its temperature. Figure 3 shows the RTD's transfer function (resistance plotted as a function of temperature) along with a linear expansion of the transfer function's slope at 0°C (for visual comparison).

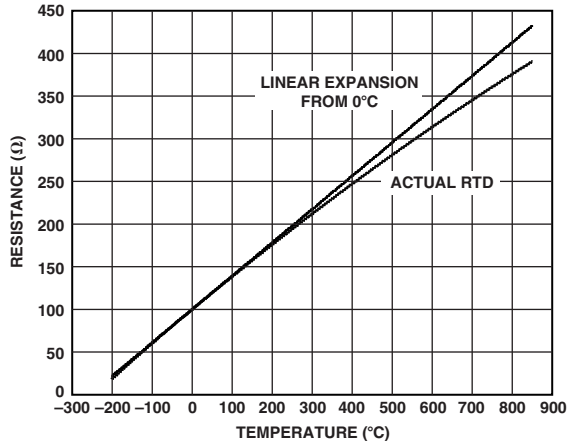

Figure 3. RTD Transfer Function

The previous equations define the RTD's resistance as a function of its temperature ($R_{RTD}(t)$). But to implement an RTD sensor interfacing circuit, the RTD's temperature must be determined instead as a function of its resistance ($T_{RTD}(r)$), which may be less straightforward, given the RTD's nonlinear transfer function. Some techniques useful for this task are explored in the following sections.

LINEARIZATION TECHNIQUES

There are a number of different ways to determine temperature as a function of RTD resistance, given the RTD's transfer function as described above. This application note examines three techniques useful in embedded designs, and more specifically, very well suited to MicroConverter-based designs. Table 1 outlines some of the strengths and weaknesses of each method, also summarizing the situations in which each might be most useful.

Table 1. Three Linearization Techniques

Technique	Advantages	Disadvantages	Summary
Direct Mathematical Method	• Very accurate • No look-up table required	• Requires math library (usually > 1 kB) • Somewhat slow (10 ms to 50 ms*)	Useful if math library is already required for other functions
Single Linear Approximation Method	• Very fast (< 1 ms*) • Very small code space requirement • Fairly accurate over narrow temperature spans • No math library required • No look-up table required	• Poor accuracy over wide temperature span	Good option when code space is limited and temperature span is fairly narrow
Piecewise Linear Approximation Method	• Fast (< 5 ms*) • Designer control of code size/accuracy trade-off • Can be very accurate • No math library required	• Greater code size than single-linear approximation method • Greater code size than mathematical method if math library is already needed for other functions	Possibly the most useful of these three methods in >90% of embedded designs

*Execution times indicated here represent empirical measurements of an ADuC834, at a core clock speed of 12.58 MHz, running the C subroutines referenced herein.

The following sections explore these linearization techniques in detail.

DIRECT MATHEMATICAL METHOD

Earlier in this document, explicit mathematical equations were shown for an RTD's resistance as a function of its temperature ($R_{RTD}(t)$). So why not just turn those equations around and solve for expressions of the RTD's temperature as a function of its resistance ($T_{RTD}(r)$)? This is a fairly straightforward task for the equation that defines positive temperature behavior, because it is merely a quadratic. The solution to the quadratic yields two expressions; to determine which one is correct, simply plug in a couple of known values. The result is the following equation for RTD temperature at temperatures of 0°C or greater:

$$T_{RTD}(r) = \frac{-A + \sqrt{A^2 - 4B\left(1 - \frac{r}{R_0}\right)}}{2B}$$

where A, B, and R_0 are defined previously (in the RTD Transfer Function section) and r is the RTD's resistance. Because this function will be solved in real time, it is beneficial to change it to the following form:

$$T_{RTD}(r) = \frac{Z_1 + \sqrt{Z_2 + Z_3 \times r}}{Z_4}$$

where:

$$Z_1 = -A = -3.9083 \times 10^{-3}$$

$$Z_2 = A^2 - 4 \times B = 17.58480889 \times 10^{-6}$$

$$Z_3 = \frac{4 \times B}{R_0} = -23.10 \times 10^{-9}$$

$$Z_4 = 2 \times B = -1.155 \times 10^{-6}$$

This is advantageous for real-time computation because Z_1 through Z_4 are constant and absolute, and so there are fewer computations to be done. The above equation for $T_{RTD}(r)$ is referred to herein as the positive function because it relates to temperatures of 0°C and above. And, because this is a direct mathematical solution, it is 100% accurate within that range. Rounding errors when using 32-bit floating-point math in 8051 C code work out to about +0.0001°C/–0.0005°C when solving this equation, which is certainly close enough to 100% accuracy for any practical purposes. Using the ADuC832 with a core clock speed of 12.58 MHz running the sample C routine of RTDmath.c, the execution time of this equation is less than 4.2 ms.

The previous equation is valid only for temperatures of 0°C and above. The equation for $R_{RTD}(t)$ that defines negative temperature behavior is a fourth-order polynomial (after expanding the third term) and is quite impractical to solve for a single expression of temperature as a function of resistance. However, making use of computer math tools can assist in finding a close approximation to the inverse transfer function. Using Mathematica® (http://www.wolfram.com/products/mathematica) or a similar software math tool, one can come up with the following best-fit polynomial expressions for RTD temperature at temperatures of 0°C or less:

$$T_{RTD}(r) = -242.02 + 2.2228 \times r + 2.5859 \times 10^{-3} \times r^2 - 4.8260 \times 10^{-6} \times r^3 - 2.8183 \times 10^{-8} \times r^4 + 1.5243 \times 10^{-10} \times r^5$$

$$T_{RTD}(r) = -241.96 + 2.2163 \times r + 2.8541 \times 10^{-3} \times r^2 - 9.9121 \times 10^{-6} \times r^3 - 1.7052 \times 10^{-8} \times r^4$$

$$T_{RTD}(r) = -242.09 + 2.2276 \times r + 2.5178 \times 10^{-3} \times r^2 - 5.8620 \times 10^{-6} \times r^3$$

$$T_{RTD}(r) = -242.97 + 2.2838 \times r + 1.4727 \times 10^{-3} \times r^2$$

These four equations are referred to herein as the negative functions because each is valid only for temperatures of 0°C and below. The top (fifth-order) equation is the most accurate but takes the longest time to compute, while the bottom (second-order) equation is the least accurate but fastest to compute. Some characteristics of these negative functions are given in Table 2, and a plot of the error of each as a function of temperature is shown in Figure 4 along with (for visual reference) the error of the positive function extended into the negative temperature space. Notice from Figure 4 that at near-zero negative temperatures, there is actually less error in the positive function than in the second-, third-, or fourth-order negative functions. The sample code RTDmath.c takes advantage of this behavior by using the positive function even at slightly negative temperatures. The actual threshold to determine if the positive or negative function should be used is different depending on which negative function (second-, third-, fourth-, or fifth-order) is used, and is represented in the Threshold column of Table 2. Above this threshold value, the positive function yields lower error; below this threshold value, the negative function yields lower error. The Equation Accuracy column of Table 2 represents errors only for temperatures below the corresponding threshold value.

Table 2. Characteristics of Best-Fit Polynomial Equations (Negative Functions)

	Maximum Execution Time*	Equation Accuracy*	Threshold
Fifth-order	41 ms	+0.0001°C/–0.00005°C	0°C/100 Ω
Fourth-order	31 ms	+0.0022°C/–0.001°C	–8.75°C/96.6 Ω
Third-order	21 ms	+0.0053°C/–0.0085°C	–12.5°C/95.1 Ω
Second-order	11 ms	+0.075°C/–0.17°C	–70.5°C/72.1 Ω

*Execution time and equation accuracy were measured empirically on an ADuC832, at a core clock speed of 12.58 MHz, running the sample C routine of RTDmath.c.

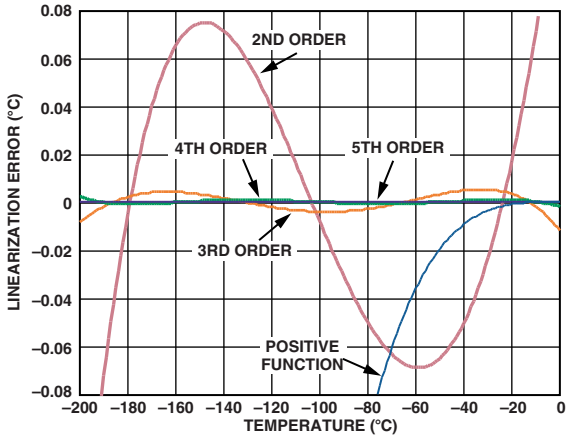

Figure 4. Error Plot of Best-Fit Polynominal Equations (Negative Functions)

One drawback of this direct mathematical technique for linearization is that it requires floating-point power and square root functions such as those found in the math library of the C51 compiler from Keil (http://www.keil.com). These floating-point math functions alone typically add more than 1 kB to the code size. Similar or better accuracy can be achieved with smaller overall code size using the piecewise linear approximation method described later in this document. However, if the math library functions are required for other operations in the program, the direct mathematical technique might be the best solution because those library functions are already available.

SINGLE LINEAR APPROXIMATION METHOD

In Figure 3, notice that over smaller temperature spans the RTD transfer function looks much like a straight line. If the required measurement temperature range spans only a portion of the full RTD measurement band, one might not need to linearize the RTD signal at all. In such cases, a best-fit linear approximation to the transfer function over the desired measurement temperature range can often yield sufficient precision. For example, over the industrial temperature range of –40°C to +85°C, a best-fit linear approximation is accurate to ±0.3°C.

In general, a linear equation for temperature as a function of RTD resistance (r) is of the form

$$T_{LIN}(r) = A \times r + B$$

where A and B are constants. Note that these are not the same A and B from the RTD Transfer Function. Choosing optimum values for A and B to minimize the error band involves some math not explored here. There is, however, a very simple software tool (intended to accompany this document) that can automatically find optimum values of A and B to fit your specific temperature range. This tool is examined later in this document, but first it must be determined whether a single linear approximation is suitable for the specific design requirement.

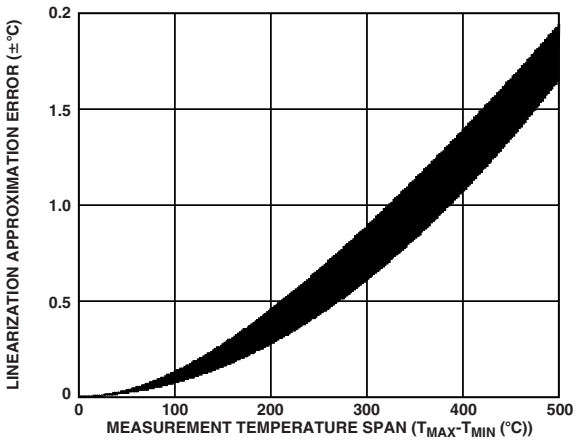

Figure 5. Single Linear Approximation Error vs. Measurement Temperature Span

Figure 5 offers a view of the total approximation error that results for measurement temperature spans of up to 500°C. For more than 500°C spans, approximation error continues to degrade with increasing temperature span. The imprecise nature of the Figure 5 plot (that is, the broad width of the data trace) is due to the fact that even for the same span of temperature, the error is different for different absolute temperature bands. For example, the temperature ranges –200°C to 0°C and +600°C to +800°C do not have the same precision even though they both span exactly 200°C. Figure 5 provides little more than a rough idea of error in order to help gauge whether single linear approximation should be considered as an option. If it is determined that it might be an option, the RTD coefficient generator tool described later can help determine the actual approximation error for a specific temperature range, and can generate source code optimized for that temperature range.

PIECEWISE LINEAR APPROXIMATION METHOD

Taking linear approximation one step further, one can conceptualize any number of linear segments strung together to better approximate the nonlinear RTD transfer function. Generating this series of linear segments so that each segment's endpoints meet those of neighboring segments results in what can be viewed as a number of points connected by straight lines. These points (or coefficients) can be calculated once to best match the RTD's nonlinear transfer function and then stored permanently in ROM or Flash memory. From this table of coefficients, the MCU can perform simple linear interpolation to determine temperature based on measured RTD resistance.

To understand how this is implemented in practice, first assume the table of coefficients already exists. Each coefficient in the table is simply a point on the transfer function, represented by a resistance and a temperature. So the table takes the form

$\{r_0,t_0;\ r_1,t_1;\ r_2,t_2;\ldots r_N,t_N\}$.

Given this table, the MCU's real-time task (in determining temperature at a given resistance r) is to first determine which two coefficients are closest to the point in question (call these $\{r_m,t_m\}$ and $\{r_n,t_n\}$), and to then linearly interpolate between those two points to solve for temperature. The actual linear interpolation formula for that range (i.e., valid only for values of *r* between r_m and r_n) will then take the form

$$T_{SEG}(r) = t_m + (r - r_m)\frac{t_n - t_m}{r_n - r_m}$$

Note that each coefficient in the above lookup table consists of two numbers, one for resistance and one for temperature (essentially x and y values in the transfer function). So for N linear segments (i.e., N+1 coefficients), a total of 2N+2 values must be stored in memory. To reduce the size of the look-up table, consider a table consisting of N segments, each spanning an equal breadth of resistance. Such a table can be stored as a set of temperature points only

$\{t_0;\ t_1;\ t_2;\ \ldots\ t_N\}$

since, for a given coefficient $\{r_n,t_n\}$, the value of r_n can be calculated by

$$r_n = r_0 + n \times r_{SEG}$$

where both r_0 and r_{SEG} are fixed values, stored in ROM along with the table of coefficients. r_0 is the resistance at coefficient zero $\{r_0,t_0\}$ and r_{SEG} is the fixed span of resistance that separates adjacent coefficients. The linear interpolation formula for a given segment then becomes

$$T_{SEG}(r) = t_i + \left[r - (r_0 + i \times r_{SEG})\right] \times \frac{t_{i+1} - t_i}{r_{SEG}}$$

where *i* indicates which segment (i.e., which pair of coefficients) is being used, and is calculated using the value of r as follows:

$$i = trunc\left(\frac{r - r_0}{r_{SEG}}\right)$$

Again, the above expression for $T_{SEG}(r)$ is nothing more than a linear interpolation between the two coefficients t_i and t_{i+1}. To implement this in practice, the MCU must first solve for *i* (per the lower of the above two equations) so that the coefficients t_i and t_{i+1} are the two closest to the input value for r. Then, with i solved for, the MCU can simply solve the equation $T_{SEG}(r)$ to determine the temperature at the given input resistance.

The overall error generated by this piecewise linear approximation technique will depend on: 1) the number of segments (or number of coefficients, or size of look-up table), and 2) the overall span of temperature. Figure 6 shows the linear approximation error for a measurement temperature range of –200°C to +850°C plotted as a function of look-up table size (using optimized coefficients generated by the RTD coefficient generator tool to be discussed shortly). Note that if the measurement temperature range is reduced, a better error will result given the same size look-up table, or the same error with a smaller look-up table.

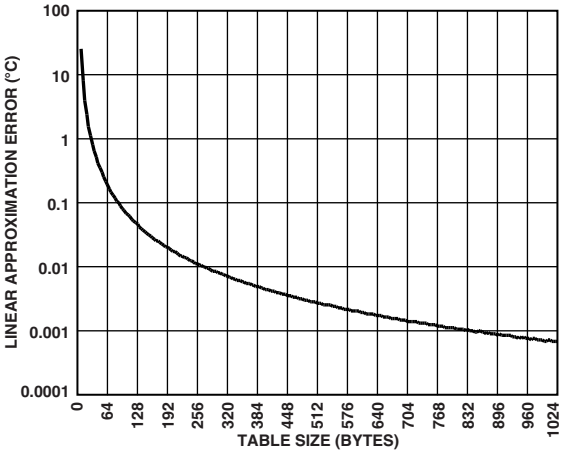

Figure 6. Piecewise Linear Approximation Error vs. Look-Up Table Size (–200°C to +850°C Range)

RTD COEFFICIENT GENERATOR TOOL

Certainly the most difficult part of implementing a piecewise linearization function is generating the look-up table. However, the RTD coefficient generator tool intended to accompany this document (coefRTD.exe) will do so automatically for Platinum RTDs. It is a simple DOS executable intended to assist with 8051-based RTD interface designs using piecewise linear or single linear approximation methods. It performs the following tasks:

- Generates optimized look-up table coefficients for a given temperature range and look-up table size.
- Indicates resulting error band and look-up table size.
- Generates complete RTD linearization function (including look-up table) in 8051 C source code.
- Generates table of error values as a function of temperature resulting from the given look-up table.

Figure 7 shows a sample screen from a typical run through the program, with user input highlighted in red. Note that the user needs to input only three parameters (T_{MIN}, T_{MAX}, and N_{SEG}) and the program does all the rest. The program can generate the file RTDpwl0.c, which is a complete C source file (customized to the user's specific look-up table) that can be included as is in a project where the T_rtd() function is available to be called directly from functions in other source files. Alternatively, any portion(s) of RTDpwl0.c can be copied and pasted directly into other source file(s). The coefficient generator can also output an error analysis file (errorRTD.txt), which is a tab-delimited text file that can be imported into Microsoft® Excel or any other spreadsheet program to examine the errors generated by the linear approximation routine.

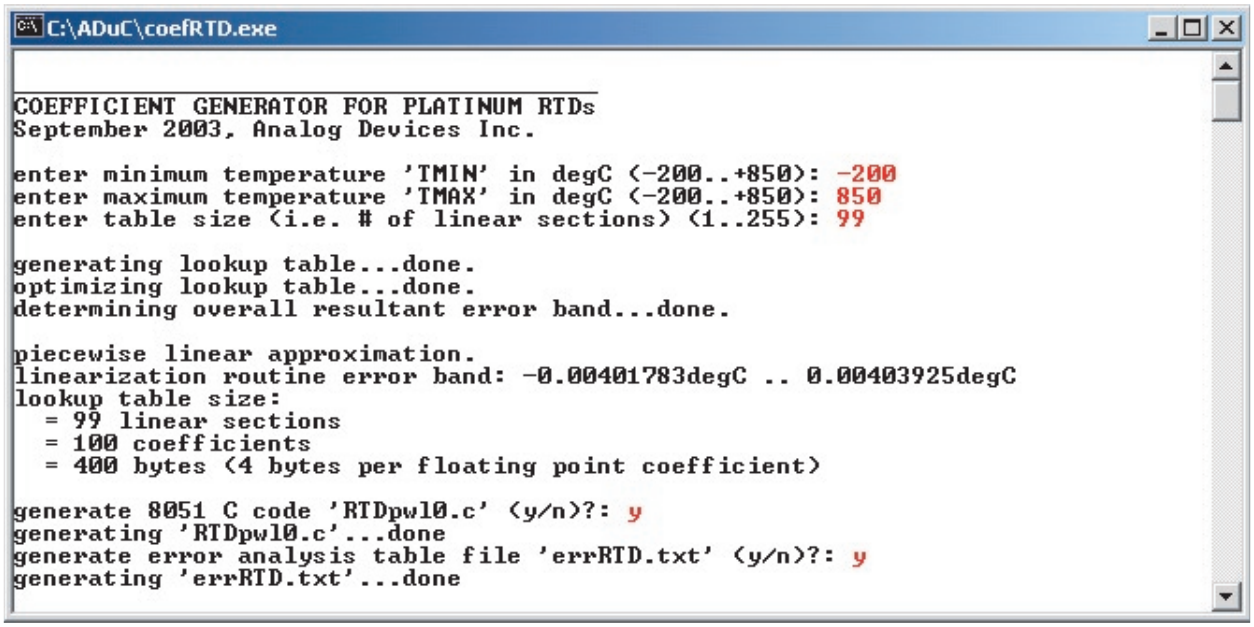

Figure 7. Coefficient Generator Session Example, with Piecewise Linear Approximation (User Input in Red)

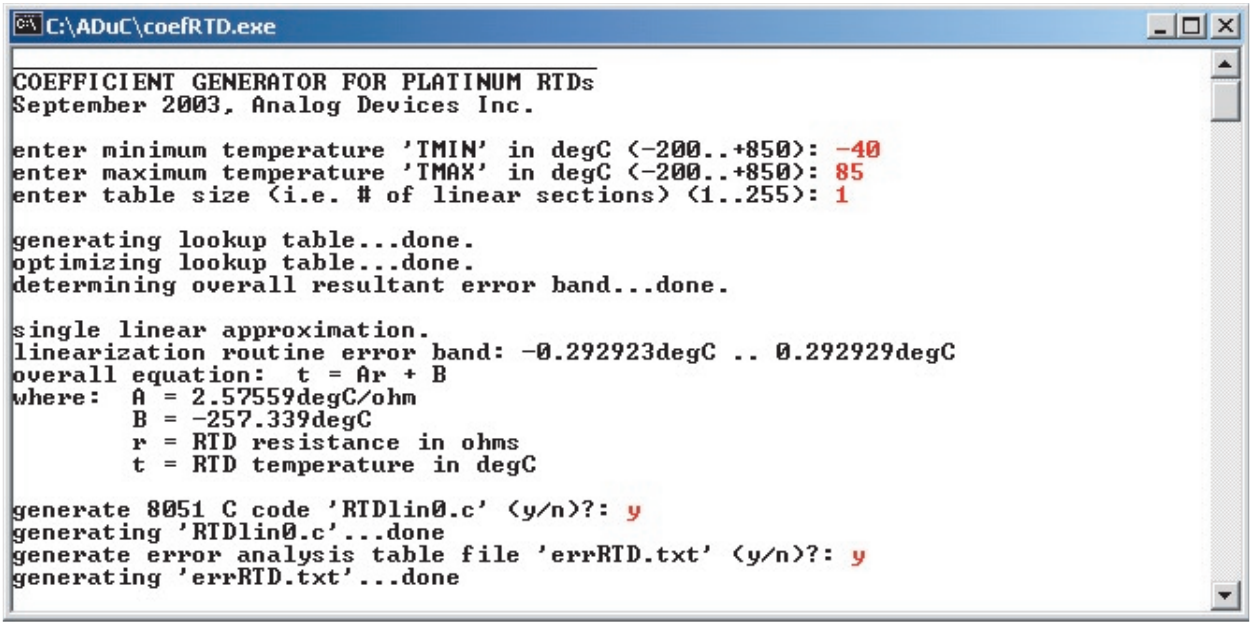

Figure 8. Coefficient Generator Session Example, with Single Linear Approximation (User Input in Red)

The coefficient generator program generates linearization functions not only for piecewise linear approximation, but also for single linear approximation. To do this, simply enter 1 for the table size to indicate only a single linear segment. The program recognizes this and outputs results pertaining to the single linear approximation method instead of the piecewise linear approximation method, as shown in Figure 8.

CALIBRATION

The ADuC834 has a built-in function to calibrate the ADC for endpoint errors (offset and gain error) as documented in the product data sheet. However, if the entire signal chain, including the RTD itself, is taken into account instead during calibration, one can end up with a lower overall error, and in such a case, the built-in ADC calibration provides no added benefit. This application note examines the overall calibration first, and then points out some instances where the built-in ADC calibration might still be useful.

Up to this point, the assumption has been that the RTD itself is perfect. But real RTDs are not perfect. Just like anything else in the real world, they have errors associated with them as specified by the RTD manufacturer's data sheet. Fortunately, much of this error can be fairly easily calibrated out in software. The calibration function discussed here works as either a single-point or a two-point calibration, and this function can be used in conjunction with any of the previously described linearization techniques.

To understand how a single-point calibration works in principle, refer back to the RTD transfer function $R_{RTD}(t)$ discussed earlier in this document and recall that it is largely defined by the value R_0, which is the resistance of the RTD at 0°C. For the most common RTDs, R_0 is nominally 100 Ω. But this R_0 value is the most significant source of error in an RTD sensor, because it can vary significantly from one device to the next. And because the R_0 value is simply multiplied by the rest of the transfer function in the expressions for $R_{RTD}(t)$, errors due to R_0 tolerance are purely multiplicative, and so can be corrected by adjusting the scale multiplier in the following expression (as given previously) for R_{RTD} as a function of normalized ADC conversion result.

$$R_{RTD} = ADC_{norm} \times scale + offset$$

Specifically, if the RTD can be brought to a very precise known temperature and an ADC conversion performed, then the corrected scale value can be calculated as

$$scale = \frac{R_{cal}}{ADC_{cal}}$$

where ADC_{cal} is the actual normalized result of the A/D conversion and R_{cal} is the ideal (expected) resistance value at that RTD temperature. R_{cal} can be calculated manually using the equations for $R_{RTD}(t)$. By this method (called single-point calibration), a corrected scale value is obtained, compensating for the RTD's R_0 tolerance and also, simultaneously, for the reference resistor's initial tolerance. To take this a step further, one can employ a two-point calibration, which compensates not only for these scaling errors but also for any offset error that might exist. Doing so requires adjusting not only the scale value, but the offset value as well. Assume for the moment that a single-point calibration has already

been performed, and the RTD can now be brought to a second very precise known temperature and another ADC conversion performed. The equation for the *scale* value (that is, the slope of the R_{RTD} versus ADC_{norm} function) is then

$$scale = \frac{R_{cal} - R_{prevcal}}{ADC_{cal} - ADC_{prevcal}}$$

where:

$R_{prevcal}$ and $ADC_{prevcal}$ are the resistance and ADC conversion result, respectively, at the previous calibration point.

R_{cal} and ADC_{cal} are the same for the current calibration point.

Note that this is merely a way of determining the slope of the R_{RTD} versus ADC_{norm} transfer function using two points on that line. Now one has only to take care of the offset value, which, because the scale value is now known, can be determined using a single point. The following expression for the offset value comes by solving for the offset of the above R_{RTD} expression and then replacing R_{RTD} and ADC_{norm} with $R_{prevcal}$ and ADC_{preval}, respectively.

$$offset = R_{prevcal} - ADC_{prevcal} \times scale$$

Note that if $R_{prevcal}$ and $ADC_{prevcal}$ are both zero (representing no prior calibration point), then the expression for the scale value becomes the same as for a single-point calibration, and the expression for the offset value becomes zero, just as if this were a single-point calibration. Therefore, the same function (Cal() in the sample code) can be used to perform either a single-point or a 2-point calibration.

If using the sample code as is, follow these steps to perform a 2-point calibration:

1. Choose two temperatures at which to perform calibration, making sure that the temperature points are sufficiently separated (ideally at least one quarter of the total measurement span) to avoid errors accumulating near the extremes of the measurement temperature range.
2. Bring the RTD to the first temperature point, wait for the displayed result to settle to the new value, and then press any key of the terminal (or terminal emulator) to bring up the user I/O menu.
3. Follow the menu prompts to calibrate to a known temperature, and then enter the temperature as prompted.
4. Repeat steps 2 and 3 for the second temperature point.

Note: for a single-point calibration, simply ignore step 4.

There are many benefits to calibration, but just as many system considerations that make it impractical for certain applications. If a calibration cannot be performed as described above, consider performing a system ADC calibration instead, as described in the ADuC834 data sheet. To do so, simply replace the RTD with a short (0 Ω) and trigger a system zero-scale calibration, and then replace the RTD with a high precision 719.36 Ω resistance and trigger a full-scale calibration. This compensates for internal ADC errors and for initial tolerance of the R_{REF} resistor, but does not account for any error of the RTD itself.

It is worth pointing out that an added benefit of the ADuC832 (and all other MicroConverter products) is that it includes nonvolatile Flash data memory on-chip, which can be used in this case to store the calibrated scale and offset values. This way, the chip can restore the calibrated values each time the system powers up, rather than requiring a calibration each time the system is powered up. The sample code RTD834.c makes use of this feature for exactly that purpose.

ERROR ANALYSIS

There are many contributing error sources to data acquisition designs (ADC linearity, input amplifier noise, resistor Johnson noise, amplifier temperature drift, resistor temperature drift, and so on). Determining which ones are dominant for a given design can be a daunting task. Fortunately, the ADuC834 integrates all the active stages into a single fully factory specified device, making error analysis a much simpler task, but one that still requires a fair bit of insight in designs such as this that involve a nonlinear sensor element. This application note, therefore, explores the few error components that happen to be most significant for the specific hardware and software configurations that have been discussed thus far in this document.

First of all, if the system is not calibrated to a specific RTD (using the single-point or 2-point calibration discussed previously), the RTD itself is almost certainly the most significant source of absolute error. This error should be well quantified on the RTD manufacturer's data sheet, and depends on the specific model of RTD chosen. This application note concentrates instead on error sources other than the RTD itself.

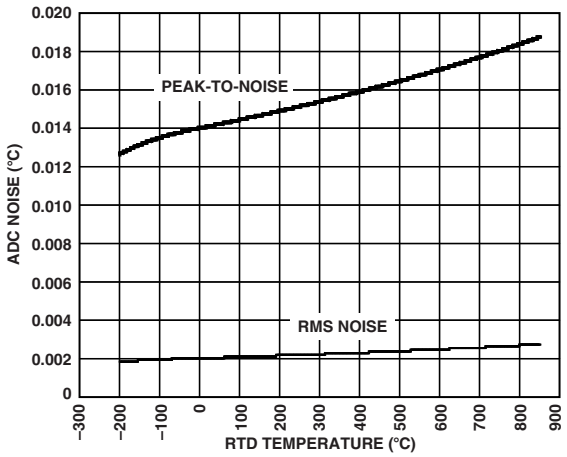

Figure 9. ADC Noise vs. RTD Temperature

One type of error to examine is noise. There are three main noise sources to consider in this design: resistor Johnson noise, amplifier/ADC input voltage noise, and amplifier/ADC input current noise. These add together as a root-sum-of-squares, and so the lesser contributing sources are negligible when one noise source is even slightly greater than the others. For this specific case, that dominant noise source happens to be amplifier/ADC input voltage noise. Specifically, at the gain setting discussed, the ADuC834's input voltage noise specification is 0.37 μV rms, or about 2.44 μV p-p. Translating this input voltage noise into the resulting output temperature noise may not be intuitively obvious and, because of the nonlinear resistance-to-temperature transfer function, results in a temperature noise that varies as a function of RTD temperature. The end result is shown in Figure 9. Notice that even at the highest RTD temperatures (that is, the worst noise), peak-to-peak noise is always below 0.019°C, and is even better at lower measurement temperatures. Keep in mind that this variation of noise as a function of RTD temperature is not a function of the ADC itself but rather, as mentioned previously, is a direct result of the nonlinear $T_{RTD}(r)$ transfer function being implemented in the digital domain.

Another source of error to consider here is temperature drift, specifically ADC offset and gain temperature drift and reference resistor temperature drift. This is the change in DC errors (offset and gain errors) as a function of changing temperature of the ADC chip or reference resistor. So this relates to ambient temperature of the RTD conditioning circuitry rather than to the actual measurement RTD temperature. For brevity, these two distinct temperatures are referred to here as ambient temperature and RTD temperature, respectively. To confuse things further, the value of temperature drift (that is, sensitivity to ambient temperature) changes as a function of RTD temperature due to the nonlinear $T_{RTD}(r)$ transfer function. The end result is shown in Figure 10 but will likely require some explanation. The x axis of Figure 10 is simply the RTD temperature. The y axis is the temperature drift in °C change in measurement error per °C change in ambient temperature. For example, if the RTD temperature is fixed at 100°C, the V_{REF} drift (with a 5 ppm/°C reference resistor) is approximately ±0.01°C/°C. So, if the ambient temperature changes by, say, 50°C, the measurement temperature reading might change by as much as ±0.5°C (neglecting other contributors to temperature drift).

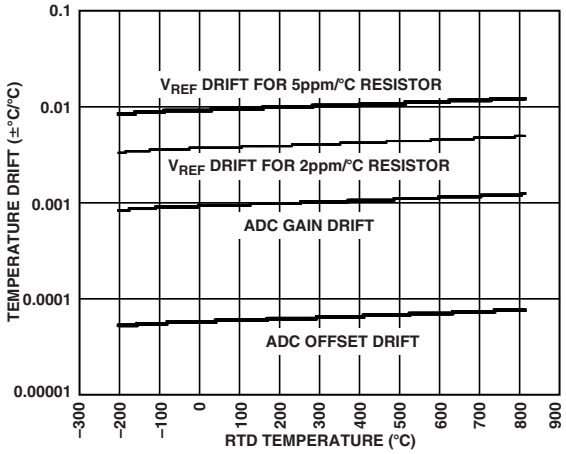

Figure 10. Temperature Drift vs. RTD Temperature

It is not difficult to see that, in industrial environments with ambient temperature ranges sometimes spanning –40°C to +85°C or more, temperature drift can be quite a significant source of error. It would be fairly straightforward to use the on-chip temperature sensor of the ADuC834 to measure chip temperature (which tracks ambient temperature closely) and then use this measured chip/ambient temperature to compensate for temperature-drift errors. This would require an additional temperature-cycling step during manufacturing, specifically bringing the ambient temperature to two fixed values and taking zero-scale and full-scale ADC readings at each of these ambient temperatures. But after doing this, software could do the rest to compensate for temperature drift errors within the limits of temperature sensor accuracy and temperature gradients between the reference resistor and the ADuC834. This application note does not explore such temperature drift compensation techniques any further, but it should be noted that the on-chip resources exist to make this option possible with nothing more than software changes.

RTD self-heating is yet another source of error to consider. Simply put, placing a current through the RTD causes it to dissipate power, which raises the RTD's temperature. Fortunately, because the RTD is being excited with only 400 μA, the total power dissipated by the RTD is never more than 16 μW for a 100 Ω R_0. The amount of self-heating caused by this small power dissipation varies, depending on the specific model of RTD used, but typically the resulting self-heating is negligible. If it is not negligible in the specific application, simply reduce the excitation current to 200 μA under software control, thus reducing RTD power dissipation by a factor of four.

Other sources of error are mostly negligible. DC endpoint errors (offset and gain errors) can be fully corrected using the calibration techniques discussed previously. Resistor Johnson noise is well below the ADC's input voltage noise. The only other error source worthy of consideration is ADC INL (integral nonlinearity, or relative accuracy). Though the ADuC834 data sheet specification for worst-case INL is 15 ppm of full scale, which would result in output-referred INL error about twice the value of the peak-to-peak output noise shown in Figure 9, the actual INL is much closer to 2 ppm of full scale, which is well below the noise floor. This is typical of ADI's conservative specifications.

SOFTWARE AND SOURCE CODE

All of the software and source code referenced herein is included in a zip file intended to accompany this document, and can be downloaded from the location mentioned on Page 1. The individual contents of the zip file can be described as follows:

- coefRTD.exe – The *coefficient generator tool* executable.
- coefRTD.cpp – Source-code for the *coefficient generator tool*.
- RTDmath.c – Linearization subroutines using direct mathematical linearization method.
- RTDpwl0.c – Linearization subroutines using piecewise linear approximation method. (A customized version of this code can be generated using the coefRTD.exe program).
- RTDlin0.c – Linearization subroutines using single linear approximation method. (A customized version of this code can be generated using the coefRTD.exe program).
- RTD834.c – Example complete RTD interface program for the ADuC834 or ADuC836. (Makes use of any one of the above linearization functions).
- RTD834.hex – Complete compiled version of RTD834.c and RTDpwl0.c, ready to download and run on an ADuC834 or ADuC836.
- RTD845.c – Example complete RTD interface program for the ADuC845, ADuC847, or ADuC848. (Makes use of any one of the above linearization functions).
- RTD845.hex – Complete compiled version of ADuC845.c and RTDpwl0.c, ready to download and run on an ADuC845, ADuC847, or ADuC848.
- ReadMe.txt – Text file giving revision information and describing the function of each file.

A complete project using the above source code must include both a main program (RTD834.c, RTD845.c, or a *from scratch* program) and a linearization subroutines file (RTDmath.c, RTDpwl0.c, RTDlin0.c, or a customized source file generated by the coefRTD.exe tool). Many details are provided in the comments of the various C source files.

AN04630–0–8/04(0)