Application Note 2024

Digital Readout and Trimming of NTC Thermistors

www.tdk-electronics.tdk.com

Digital Readout and Trimming of NTC Thermistors

Introduction

The combination of low power consumption, high sensitivity, and signal stability makes NTC (negative temperature coefficient) thermistors the most popular temperature sensor choice in automotive battery management, motor and climate control as well as factory automation and field instrumentation.

In this application note, the basic circuit design considerations will be explained to convert the resistance change of the NTC into a digital temperature readout. The example circuit uses the ADS1115, a 16-bit delta-sigma ADC from Texas Instruments (TI), to convert the voltage drop of a K560 surface sensor from TDK Electronics to a 16-bit I^2C output for skin temperature sensing.

Alternative resistance to temperature calculations will be compared: exponential curves, lookup tables and Steinhart-Hart equation. For all cases, Python-3 code or C++ code is available for download, which can be adapted for other applications and other NTC curves in own projects.

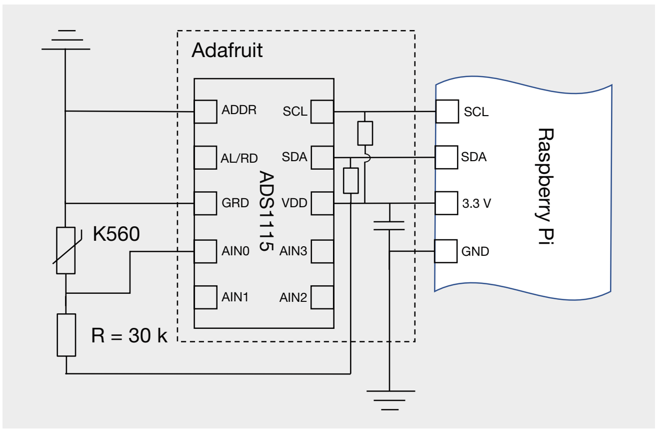
The Python and C++ class definitions for NTC thermistors enable developers to calculate temperatures from resistance readings and vice versa. In addition, a class function is available to apply single-point calibration based on an own measuring point or NTC with single-calibration data like the new B57868S0202H. The different classes are available for download.

Table of contents

Digital Readout and Trimming of NTC Thermistors

1. Hardware setup and circuit

Figure 1 shows the example circuit of the development set-up: An ADS1115 ADC from TI [1] is used on a ready to use module from Adafruit [2] which has all necessary passives on board. The ADC is connected to a Raspberry Pi that provides the 3.3 V voltage supply and the I²C interface. The key component for the circuit is a surface temperature sensor K560 from TDK [3]. Together with the fixed resistor R = 30 kΩ the NTC forms a voltage divider to provide the analog input for the ADC. K560 is originally designed for the temperature control of hot plates and induction hobs from +100 °C to +250 °C.



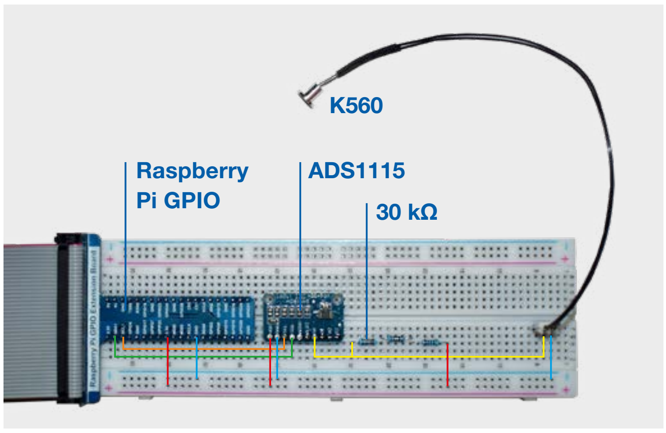


Figure 1: Evaluation circuit and breadboard wiring using a K560 thermistor probe from TDK for surface temperature sensing, an ADS1115 from TI as 16-bit ADC connected to a Raspberry Pi 3 via I²C.

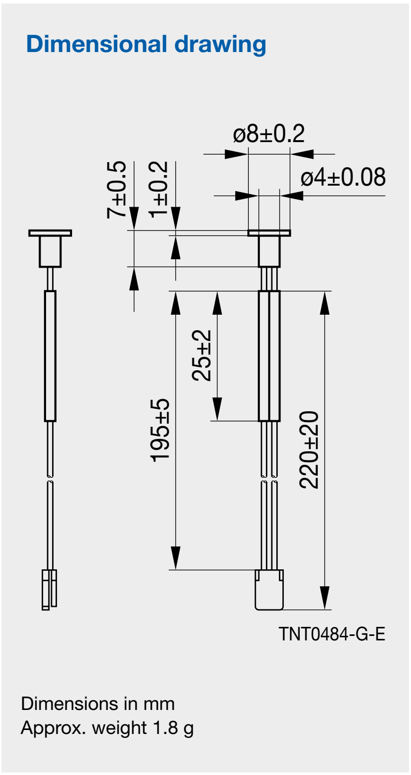
In this application example the K560 will be applied to measure skin temperature from +30 °C to +45 °C with an accuracy target of 0.1 K. The high basic resistance of 49.12 kΩ at 25 °C and the unique shape of the aluminum packaged head make this NTC an excellent choice for remote surface temperature sensing (see Figure 2). The typical sensitivity of α = -4%/K is about ten times higher than those of metals and about five times higher than those of silicon temperature sensors.

With a PGA setting of "1" the ADS1115 maps a voltage drop of 4.096 V to a single ended output of 32767 (15 bit) and the 3.3 V supply covers about 26400 counts. The bit-reading "Out" from the ADS1115 converts to a voltage drop or a NTC resistance by the following equations:

$$\frac{\text{Out}}{26400} = \frac{V_{NTC}}{3.3\text{ V}} = \frac{R_{NTC}}{R_{NTC} + R}$$

Other values for the fixed resistor and the gain can be used to optimize the placement of the measuring range within the output range. With the given settings the NTC resistance can be calculated by:

$$R_{NTC} = R \cdot \frac{V_{NTC}}{3.3\text{ V} - V_{NTC}} = R \cdot \frac{\text{Out}}{26400 - \text{Out}}$$



R_{100} Ω	R_{25} Ω	$B_{25/100}$ K	$B_{0/100}$ K
3300	49120	4006	3970 ±2%

Figure 2: K560 surface temperature probe from TDK. Drawing and values taken from [3]

Digital Readout and Trimming of NTC Thermistors

2. Resistance-to-temperature conversion

The dependence of the NTC resistance on temperature is usually approximated by the following exponential equation. Note that all temperatures need to be converted to absolute Kelvin scale for this calculation:

$$R(T) = R_R \cdot e^{B \cdot \left(\frac{1}{T} - \frac{1}{T_R}\right)} \tag{1}$$

The B-value is defined by two reference values of temperature and resistance:

$$B_{T1/T2} = \frac{T_2 \cdot T_1}{T_2 - T_1} \ln\left(\frac{R_1}{R_2}\right) \tag{2}$$

In case of the K560, the rated temperature T_R = +100 °C is used together with the 0 °C value to define the B0/100-value on the data sheet (see Figure 2):

$$B_{0/100} = 1019.3\ \text{K} \ln\left(\frac{162.213\ \text{k}\Omega}{3.3\ \text{k}\Omega}\right) = 3970.16\ \text{K} \tag{3}$$

The datasheet B value together with the rated resistance R_R and rated temperature T_R can be used to write a software code for the resistance to temperature conversion. Figure 3 shows the code example of a very basic python class definition based on the inversion of equation 1:

$$\frac{1}{T} = \frac{1}{T_R} + \frac{1}{B} \ln\left(\frac{R}{R_R}\right) \tag{4}$$

An instance of the class is initiated with the rated temperature, the rated resistance and the B-value. The resistance to temperature conversion can be done with the class function “temperature”. The following code lines will produce “36.3262” as output to the screen.

```
from ntc import NTC_B
k560=NTC_B(100, 3300, 3970)
print('%.4f'%k560.temperature(29456))
```

```
##############################################################################
class NTC_B():
##############################################################################
# The basic NTC class using rated temperature, resistance and B-value
# t_val - rated temperature in Celsius
# r_val - rated resistance in kOhm
# b_val - B-Value of Thermistor in K
# temperature() - Converts a resistance res to a temperature in Celsius
##############################################################################
    def __init__(self, t_val, r_val, b_val):
        self.b = b_val          #B-value
        self.r_r = r_val        #rated resistance
        self.t_r = t_val        #rated temperature

    def temperature(self,res):
        out = math.log(res/self.r_r) / self.b + 1 / (273.15 + self.t_r)
        out = 1 / out - 273.15
        return out
##############################################################################
#end of class NTC_B
##############################################################################
```

Figure 3: The very basic NTC Python class using rated temperature, resistance, and B-value to convert resistance read to temperature output in Celsius

Digital Readout and Trimming of NTC Thermistors

2. Resistance to temperature conversion

2.1 Formula error using the NTC equation

The approach based on data sheet values of rated resistance and temperature and B-value is only suitable for a restricted range around the rated temperature T_R with sufficient accuracy. Figure 4 shows the difference between the actual temperature and the calculated temperature based on equation 1 that occurs if $B_{0/100}$ = 3750 K is used together with the rated temperature of +100 °C and the rated resistance of 3.3 kΩ to initiate the NTC class. From +20 °C to +45 °C the calculated temperature deviates from the actual value by more than 0.5 K as the rated temperature of +100 °C is too far away from the range of use.

For practical applications a more precise software modeling of the real R_{nom}(T) curve is required. In consequence as a first step we have to get the real R/T curves! This can be quite a challenge as many manufacturers only provide such data via their direct customer support.

TDK has the real R_{nom}(T) curves available online!
In this case, the TDK online database was used to generate a detailed table for the specific use case of skin temperature sensing between +30 °C and +45 °C as shown in Table 1: The real physical values of R_{nom} will be used in sections 2.2 and 2.3 to improve the resistance to temperature conversion. α is the relative sensitivity defined by the relative change of resistance per temperature interval:

$$\alpha = \frac{1}{R_{NTC}} \frac{\partial R_{NTC}}{\partial T} \quad (5)$$

Section 2.4 shows how α values can be used to interpolate lookup table values. R_{min} and R_{max} describe the manufacturing batch variance. Section 3 explains how to deal with such batch variance by an individual calibration (“software trimming”).

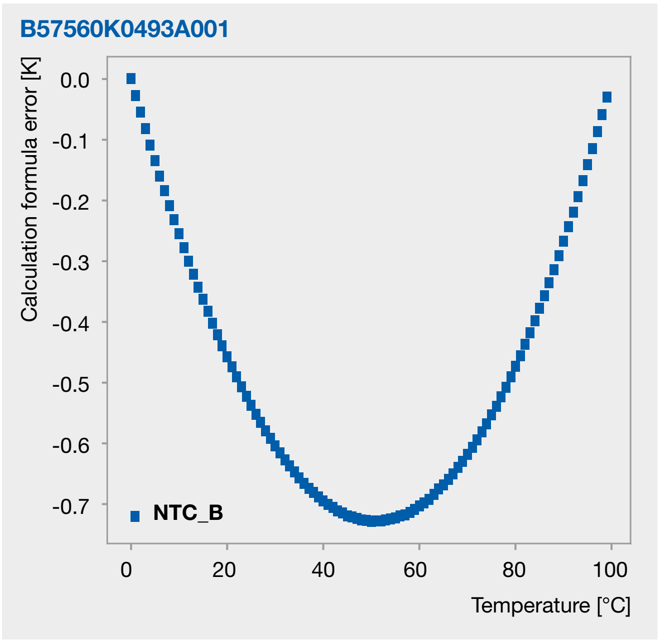


Figure 4: Difference between actual temperature and calculated temperature based on equation (1)

T [°C]	R_{nom} [kΩ]	R_{min} [kΩ]	R_{max} [kΩ]	α [%/K]
30	39.517	36.489	42.545	4.3
31	37.864	34.998	40.730	4.3
32	36.290	33.576	39.003	4.2
33	34.789	32.220	37.359	4.2
34	33.359	30.926	35.793	4.2
35	31.996	29.690	34.301	4.2
36	30.696	28.511	32.880	4.1
37	29.456	27.386	31.525	4.1
38	28.272	26.310	30.234	4.1
39	27.143	25.283	29.003	4.1
40	26.065	24.301	27.828	4.0
41	25.035	23.363	26.708	4.0
42	24.052	22.466	25.638	4.0
43	23.113	21.609	24.617	4.0
44	22.215	20.788	23.643	3.9
45	21.358	20.003	22.712	3.9

Table 1: Detailed R/T- values of K560 extracted from the TDK online database [4]

2. Resistance to temperature conversion

2.2 Algorithm based on a two-point calibration

The main drawback of the basic class function shown in Figure 3 is the use of a B-value which does not fit the desired application range. Calculating a more appropriate B-value within the actual application range using Equation $\boxed{2}$ can result in a significant improvement.

Figure 5 shows an example code that includes the B-value calculation. The Python class "NTC 2P" receives a pair of temperature and resistance values [T_1;R_1] and [T_2;R_2] to calculate an interval-specific B-value upon invoking an instance of the class.

In this particular case two data points for +30 °C and +40 °C are a good choice. The following code lines will produce "36.9932" as temperature output.

from ntc **import** NTC_2P
k560=NTC_2P([[30, 39517], [40, 26065]])print('%.4f'%k560.
temperature(29456))

```
##############################################################################
class NTC_2P():
##############################################################################
# NTC class using two data points [T,R] to calculate a B-Value.
# The first point is used to set the rated temperature t_r and resistance r_r
# data_tr  - Array of 2 datapoints [T,R]
# temperature() - Converts a resistance res to a temperature in Celsius
##############################################################################
    def __init__(self,data_tr):
        # Calculate the B-value inbetween the given datapoints
        self.b = (data_tr[0][0] + 273.15) * (data_tr[1][0] + 273.15)
        self.b = self.b / (data_tr[0][0] - data_tr[1][0])
        self.b = self.b * math.log(data_tr[1][1] / data_tr[0][1])
        self.t_r = data_tr[0][0]  # set rated temperature
        self.r_r = data_tr[0][1]  # set rated resistance

    def temperature(self,res):
        out = math.log(res / self.r_r) / self.b + 1 / (273.15 + self.t_r)
        out = 1 / out - 273.15
        return out
##############################################################################
#end of class NTC_2P
##############################################################################
```

Figure 5: NTC class using two data points [T_i; R_i] to calculate a use case specific B-value

2. Resistance to temperature conversion

2.3 Algorithm based on Steinhart-Hart equation

In 1968, John S. Steinhart and Stanley R. Hart published a higher order approach for the relation between temperature and NTC resistance [5].

$$\frac{1}{T} = C_0 + C_1 \ln(R) + C_3 \ln(R)^3 \qquad (6)$$

With three data points from the actual R/T table (1) the coefficients can be calculated by solving the following linear equation:

$$\begin{pmatrix} 1/T_1 \\ 1/T_2 \\ 1/T_3 \end{pmatrix} = \begin{pmatrix} 1 & \ln(R_1) & \ln(R_1)^3 \\ 1 & \ln(R_2) & \ln(R_2)^3 \\ 1 & \ln(R_3) & \ln(R_3)^3 \end{pmatrix} \cdot \begin{pmatrix} C_0 \\ C_1 \\ C_3 \end{pmatrix} \qquad (7)$$

The code example in Figure 6 uses Python’s numerical package NumPy to calculate the coefficients from three data points of the R/T curve. In the code package which is provided for download at TDK NTC design tool pages, an explicit calculation formula is used to avoid NumPy and to make the code usable for CircuitPython platforms. C++ implementations are also available for download.

The temperature ( ) function uses equation 6 to calculate the output. For the application example of skin temperature sensing, the resistance data at +30 °C, +35 °C and +40 °C was used as input to the class to calculate the Steinhart-Hart coefficient. The following lines will produce “36.9997” as temperature output:

```
from ntc import NTC_SH
k560 = NTC_SH([[30, 39517] ,
[35, 31996] ,[40, 26065]])
print(‘%.4f’%k560.temperature(29456)
```

```
##############################################################################
class NTC_SH():
##############################################################################
# NTC class using three data points [T,R] to calculate the Steinhart Hart
# coefficients. numpy is required for the matrix calculations!
# data_tr  - Array of 3 datapoints [T,R]
# temperature() - Returns a temperature in C as function of resistance
##############################################################################

    def __init__(self,data_tr):
        # for recalibration original data_tr must be known:
        self.data_tr = data_tr
        # use inv_t=1/T as internal variable in place of T
        inv_t = np.array([1 / (data_tr[0][0] + 273.15),
                         1  /(data_tr[1][0] + 273.15),
                         1  /(data_tr[2][0] + 273.15)])
        # use ln_r=ln(R) as internal variable in place of R
        ln_r = np.array([[1,1,1],
                        [math.log(data_tr[0][1]),
                        math.log(data_tr[1][1]),
                        math.log(data_tr[2][1])],
                        [math.log(data_tr[0][1]) ** 3,
                        math.log(data_tr[1][1]) ** 3,
                        math.log(data_tr[2][1]) ** 3]])
        # calculate the Steinhart Hart coefficients
        self.sh = np.matmul(inv_t, np.linalg.inv(ln_r))

    def temperature(self,res):
        out = self.sh[0] + self.sh[1] * math.log(res)
        out = out + self.sh[2] * math.log(res) ** 3
        out= 1 / out - 273.15
        return out
##############################################################################
#end of class NTC_SH
##############################################################################
```

Figure 6: NTC class using three data points [Ti ; Ri] to calculate the Steinhart-Hart coefficients. NumPy is required for the matrix calculations!

2. Resistance to temperature conversion

2.4 Discussion of formula error and lookup tables

The models we discussed so far are based on 2 -point NTC_2P or 3-point NTC_SH) models of the true NTC curve. For wider temperature ranges or depending on the math-capabilities and the available memory of the given controller the use of look-up tables can be an advantage. Instead of two or three points, an algorithm is built upon multiple [Ti;Ri] points, e.g. with 5 K steps or even 1 K steps within the operation range. For resistance readings in between two data points Ri and Ri+1 the T-values can be extrapolated. For a very detailed R/T table even a linear interpolation might be suitable:

$$T(R) = T_i + \frac{T_{i+1} - T_i}{R_{i+1} - R_i} \cdot (R - R_i) \quad ; \quad R_i \leq R < R_{i+1} \tag{8}$$

As an alternative for fewer input points (i.e. 5 K steps) the α-values 5 from the datasheet or from the TDK online database can be used. Please note that the α-value is related to the temperature dependent $B_{Ti/Ti+1}$-value within each interval by:

$$B_{T_i/T_{i+1}} = -\alpha_i T_i^2 \quad ; \quad B_{T_i/T_{i+1}} = \frac{T_{i+1} \cdot T_i}{T_{i+1} - T_i} \ln\left(\frac{R_i}{R_{i+1}}\right) \tag{9}$$

For a resistance output between Ri and Ri+1 the temperature calculation can be done by:

$$\frac{1}{T} = \frac{1}{T_i} + \frac{1}{B_{T_i/T_{i+1}}} \ln\left(\frac{R}{R_i}\right) \tag{10}$$

The additional class definition is available for download at the TDK NTC design tool pages. The class NTC_LT() uses a list of [Ri ,Ti] values and equation 9 and 10 to interpolate between the list values. Figure 7 shows a comparison of the calculation error for all three models. The NTC_2P() creates a parabolic residual calculation error with maximum 0.05 K deviation in the range +25 °C and +45 °C which is already sufficient for many applications. Also, the look up table algorithm used by the class NTC_LT() shows the parabolic error profile in between each of the supporting points. The Steinhart-Hart model as a higher-order approximation provides the lowest calculation error but requires more complicated calculation.

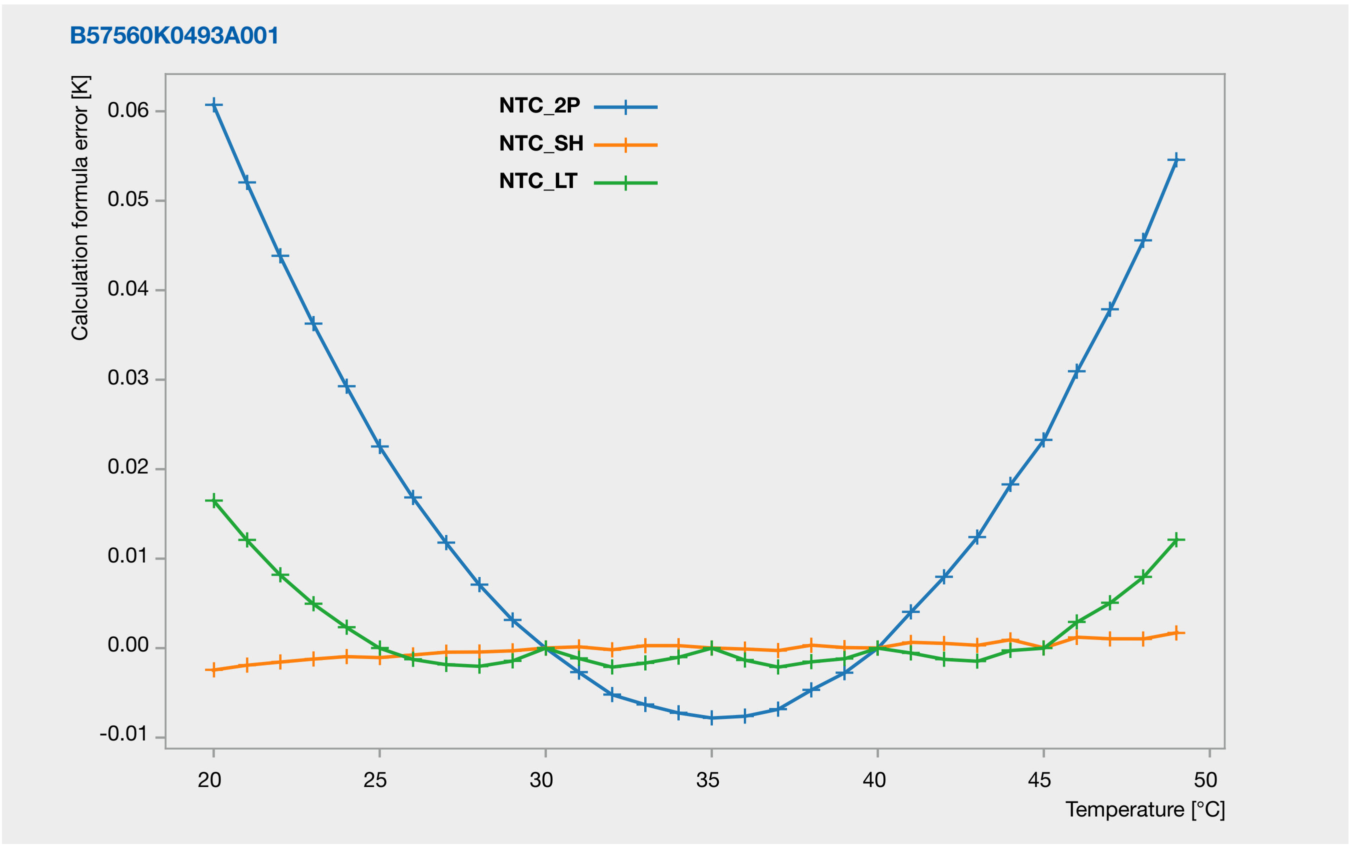


Figure 7: Difference between actual temperature and calculated temperature from the three different algorithms

Digital Readout and Trimming of NTC Thermistors

3. Software-based trimming

The main drawback in using a temperature probe that was designed for induction hobs as skin temperature probe is the manufacturing tolerance (R_{min}, R_{max} in R/T table 1). Manufacturing tolerance means that the resistance reading of one specific probe might deviate from the nominal value but remains within the minimum to maximum limits. In the case of K560 the rated resistance R_R of 3.3 kΩ has a manufacturing tolerance of 2.5% and in consequence a temperature tolerance of 0.9 K at T_R = 100 °C.

However, at +36 °C there is already a temperature formula error of:

$$\Delta T = \frac{1}{\alpha} \frac{\Delta R}{R} = \pm 1.7\ \text{K}$$

For the algorithm in sections 2.2 and 2.3, two and three data points were used to calculate the relevant coefficients for equations 4 and 6 respectively. One might now assume that in consequence, two or three measured points are necessary as well to eliminate the manufacturing variance. However, this is not the case. With a single measurement most of the manufacturing errors can be eliminated. This will be demonstrated with the two-point method from section 2.2.

The relative resistance tolerance from manufacturing can be split into contributions from R_N and B:

$$\left|\frac{\Delta R}{R}\right| = \left|\frac{\Delta R_R}{R_R}\right| + \left|\frac{\Delta_B R}{R}\right| = \left|\frac{\Delta R_R}{R_R}\right| + \left|\ \Delta B \cdot \left(\frac{1}{T_R} - \frac{1}{T}\right)\right|$$

As the NTC_2P() class from section 2.2 calculated a $B_{30/40}$-value and we used the reference point $[T_R ; R_R]$ = [30 °C, 39517 Ω] as rated resistance, the contribution of B is very small as it scales with $\left(\frac{1}{T_R} - \frac{1}{T}\right)$:

$$\Delta_B T = \frac{1}{\alpha} \left|\Delta B \cdot \left(\frac{1}{T_R} - \frac{1}{T}\right)\right| = \frac{1}{4\%/\text{K}}$$

$$2\% \cdot 3950\ \text{K} \cdot \left(\frac{1}{303{,}15\ \text{K}} - \frac{1}{306{,}15}\ \text{K}\right)\Big| \leq \pm 0.12\ \text{K}$$

The idea therefore is to eliminate $\left|\frac{\Delta R_R}{R_R}\right|$ by a single-point measurement and reduce the manufacturing tolerance impact by more than factor 10 from 1.7 K to 0.12 K.

It is easy to extend the code class NTC_2P() by two more functions to enable calibration. The code for “resistance()” and “calibration()” is shown in the listing of Figure 8.

The idea is to give the result of an actual temperature and resistance measurement as input and change the class parameters in a way, that the actual measured resistance will become rated resistance and the actual measured temperature will be the new rated temperature. Of course, the contribution of ΔR_N is still important but will depend only on the accuracy of the one measurement and not on manufacturing variance anymore.

In the class NTC_2P() only the self.r_r parameter needs to be changed by the factor $R_{measured}$ /$R(T_{measured})$, where $R(T_{measured})$ is the calculated resistance based on the old parameters. In the case of the Steinhart-Hart approach in the code class NTC_SH() the coding is more complicated, as the inversion of equation 6 requires Cardan’s method and a full recalculation of all Steinhart-Hart coefficients. It is not re-printed here but TDK made it available as part of the download package at the TDK NTC design tool pages.

```
def resistance(self, tem):
# calculates the resistance from a given temperature tem in Celsius
    out = self.b * (1 / (tem + 273.15)-1 / (self.t_r + 273.15))
    return self.r_r * math.exp(out)

def calibrate(self, point_tr):
# point_tr is a data point of [T(°C),R] used for calibration
    factor = point_tr[1] / self.resistance(point_tr[0])
    self.r_r = self.r_r * factor
```

Figure 8: Extension for the Python class from section 2.2 to enable one-point trimming

For the calibration measurement it is important that sensor and reference reach thermal equilibrium. Figure 9 shows how a stable resistance is reached after approx 1 minute. As a reference a commercial fever sensor was used. The resistance reading of the thermistor (30456 Ω) and the temperature reading of the reference (+36,4 °C) will be used for calibration.

After soft trimming with the following code lines, the K560 circuit output will be in line with the reference:

```
from ntc import NTC_2P
k560=NTC_2P([[30, 39517], [40, 26065]])
k560.calibrate(36.4, 30456)
```

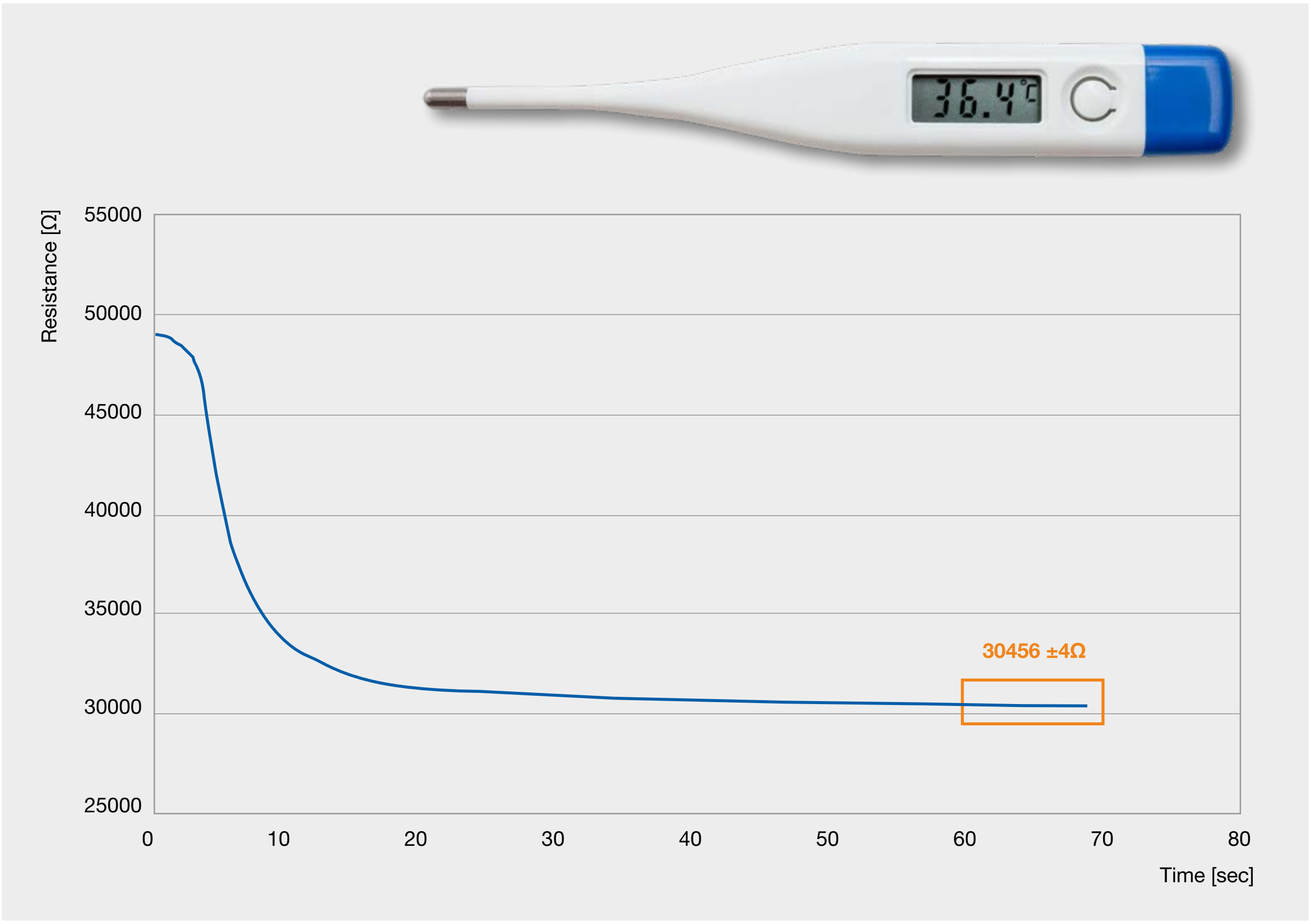


Figure 9: Example time plot for a calibration measurement

To make single calibration even easier, TDK offers NTC types with individual resistance data such as the B57868S0202H type [6]. Actual measured resistance values for the rated temperature are available for each part as CSV file. This S868 type has a rated temperature of +100 °C and a rated resistance of 181.55 Ω. The nominal resistance value at +25 °C equals 2000 Ω. Following the example in the datasheet [6] the 7th part on a delivery strip has an actual measured resistance at +100 °C of R_{Meas} = 181.92 Ω and a correction R-factor of 1.00204. Single calibration on software level can be done with the following code:

```
from ntc import NTC_2P
s868_7=NTC([[25, 2000], [100, 181.55]])
s868_7.calibrate(100, 181.92)
```

As an alternative, all necessary resistance values of the nominal curve can be multiplied by the R-factor (1.00204) and used directly to initiate the class. One might find this method more convenient, especially for the NTC_SH and NTC _LT classes:

```
rfactor_7=1.00204
from ntc import NTC_2P
s868_7=NTC([[25, 2000.0 * rfactor_7], [100, 181.55 * rfactor_7]])
```

4. Conclusion

NTC thermistor sensors and probes can easily be integrated in digital circuits especially for remote sensing.

Signal stability and low power consumption allow an easy circuit design. The use of actual temperature vs. resistance tables allows for more exible and accurate software algorithms than data sheet B-values and rated resistance values. Manufacturing variance can be reduced by a factor of 10 with single-point measurement and software based trimming. Python 3 and C++ implementations for all classes were provided for download at the TDK NTC design tool pages.

References

[1] **Texas Intruments**
ADS111x ultra-small, low-power, I^{2}C-compatible, 860-SPS, 16-Bit ADCs
With internal reference, oscillator, and programmable comparatoor https://www.ti.com/product/ADS1115, 2018.

[2] **Adafruit Industries**
Adafruit 4-channel ADC breakouts
https://learn.adafruit.com/adafruit-4-channel-adc-break-outs/downloads, 2020.

[3] **TDK Electronics AG**
NTC probe assembly K560
https://www.tdk-electronics.tdk.com/inf/50/db/ntc/NTC_Probe_ass_K560.pdf, 2018 edition.

[4] **TDK Electronics AG**
www.tdk-electronics.tdk.com/web/designtool/ntc/

[5] **John S. Steinhart and Stanley R. Hart**
Calibration curves for thermistors
Deep sea research and oceanographic abstracts, 15(4):497-503, 1968.

[6] **TDK Electronics AG**
Datasheet of B57868S0202H NTC thermistor

1.) NTC: A simple B-value based calculation and a two-point definition of R/T curves

2.) NTC_SH: The Steinhart-Hart model with three data points

3.) NTC_LT: a lookup table class using multiple data points

All class definitions include a class function for a single point software trimming of a given NTC. One code example is included that shows, how the csv output from NTC R/T calculation 5.0 - web-based application can be used to define the classes for a specific NTC. A second code example demonstrates the class usage in a typical circuit with NTC and analog to a digital converter. All class definitions include a class function for a single point software calibration to eliminate manufacturing tolerances.

Download the software classes tool here

DOWNLOAD

Address and contact info
Dr. Bernhard Ostrick
Product Development
Temperature & Pressure Sensors Business Group
bernhard.ostrick@tdk.com
www.tdk-electronics.tdk.com

© TDK Electronics AG
Edition 03/2024
www.tdk-electronics.tdk.com