TDK

Digital readout and trimming of NTC thermistors

The combination of low power consumption, high sensitivity and signal stability makes NTC thermistors the most popular temperature sensor choice in automotive battery management, motor and climate control as well as factory automation and field instruments.
In this application note the basic circuit design considerations will be explained to convert the NTC's resistance change into a digital temperature readout. The circuit example uses an ADS1115 from TI to convert the voltage drop of a K560 surface sensor to a 16 bit I2C 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 is available for download that can be adapted for other applications and other NTC curves in own projects using Python 3 or CircuitPython.

The Python class definitions for NTC thermistors enable developers to calculate temperatures from resistance readings and vice versa. The different classes are available and can be downloaded here.

1. Hardware setup and circuit

Figure 1 shows the example circuit of my development set-up: I am using an ADS1115 ADC from TI [1] on a ready to use module from Adafruit [2] that 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 at 100 °C to 250 °C.

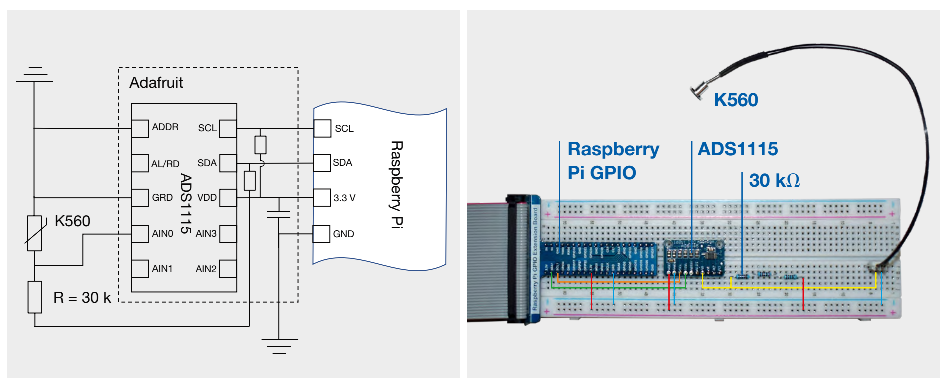


Figure 1: Evaluation circuit and breadboard wiring using a K560 thermistor probe from TDK for surface temperature sensing, an ADS1115 from TI as 16 Bit ADC connected to a Raspberry Pi 3 via I2C.

© TDK Electronics AG · Edition 07/21

Contents

1. Hardware setup and circuit ... 1

2. Resistance to temperature conversion ... 3

2.1 Formula error using the NTC equation ... 3

2.2 Algorithm based on a 2 point calibration ... 4

2.3 Algorithm based on Steinhart-Hart equation ... 5

2.4 Discussion of formula error and lookup tables ... 5

3. Software based trimming ... 6

4. Conclusion ... 8

References ... 8

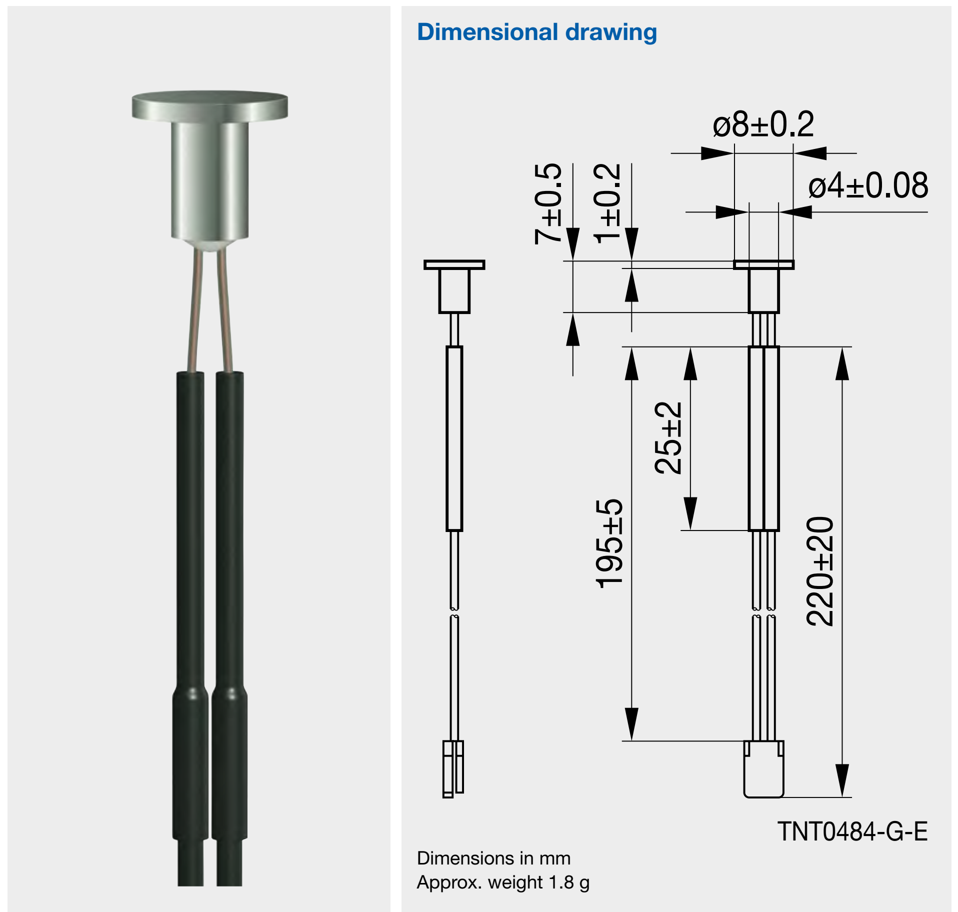
In this application example the K560 will be applied to measure skin temperature between 30 °C to 45 °C with an accuracy target of 0.1 K. The high basic resistance of 49.12 kΩ at 25 °C and the unique shape of the aluminum packaged head make it an excellent choice for remote surface temperature sensing (see figure 2). NTC (Negative Temperature Coefficient) thermistors are thermally sensitive semiconductor resistors which show a decrease in resistance as temperature increases. With a typical sensitivity of α = -4%/K, the sensitivity is about ten times greater than those of metals and about five times greater than those of silicon temperature sensors. We come back to the temperature dependence of the NTC in section 2.

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. The bit-reading “Out” from the ADS1115 converts to a voltage drop or a NTC resistance by the following equations:

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

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:

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



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]

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 calculation:

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

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

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

In case of K560 the rated temperature T_R = 100 °C is used together with the 0 °C value to define the $B_{0/100}$-value on the data sheet (see figure 2):

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

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:

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

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 would 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

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.

Between 20 °C to 45 °C the calculated temperature deviates from the actual value by more than 0.5 °C as the rated temperature of 100 °C is too far away from the range of use.

For practical applications a more precise modelling 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.

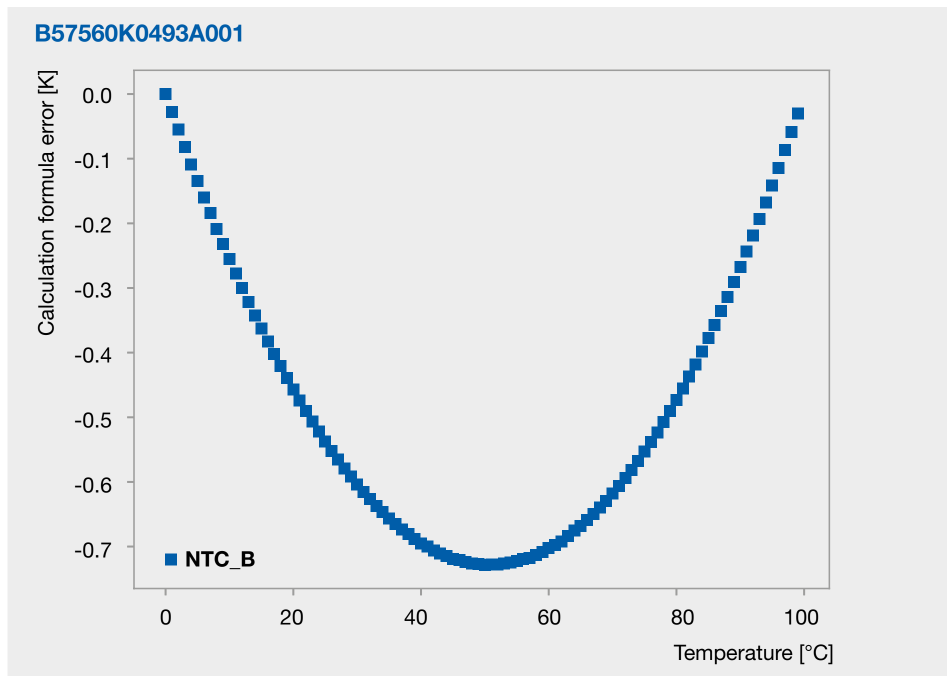


Figure 4: Difference between actual temperature and calculated temperature based on equation (1)

TDK has them all available online: I used the TDK online database 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:

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

in section 2.4 it will be described how alpha values can be used to interpolate lookup table values. R_{min} and R_{max}, the manufacturing batch variance. In section 3 it will be described how to deal with such batch variance by an individual calibration.

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]

2.2 Algorithm based on a two point calibration

The main drawback of the basic class function in figure 3 is the use of a B-values that does not fit the desired application range. A significant improvement can be made by calculating a more suitable B-value within the actual application range by equation 2.

Figure 5 shows an example code that includes the B-value calculation. The Python class 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 my use 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

2.3 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].

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

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

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

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 that we provided for download at TDK NTC design tool pages we used an explicit calculation formula to avoid numpy and to make the code usable for CircuitPython platforms.

The temperature ( ) function uses equation 6 to calculate the output. For the application example of skin temperature sensing I used the resistance data at 30 °C, 35 °C and 40 °C 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 datapoints [T_i; R_i] to calculate the Steinhart-Hart coefficients. numpy is required for the matrix calculations!

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 build upon multiple [Ti;Ri] points e.g. with 5K steps or even 1K 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:

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

As an alternative for less 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 $B_{T_i/T_{i+1}}$-value within each interval by:

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

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

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

An additional Python class definition is available for download at the TDK NTC design tool pages.

The class NTC_LT() uses a list of $[R_i,T_i]$ values and equation 9 and 10 to interpolate in 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 higher order approximation provides the lowest calculation error but requires more complicated calculation.

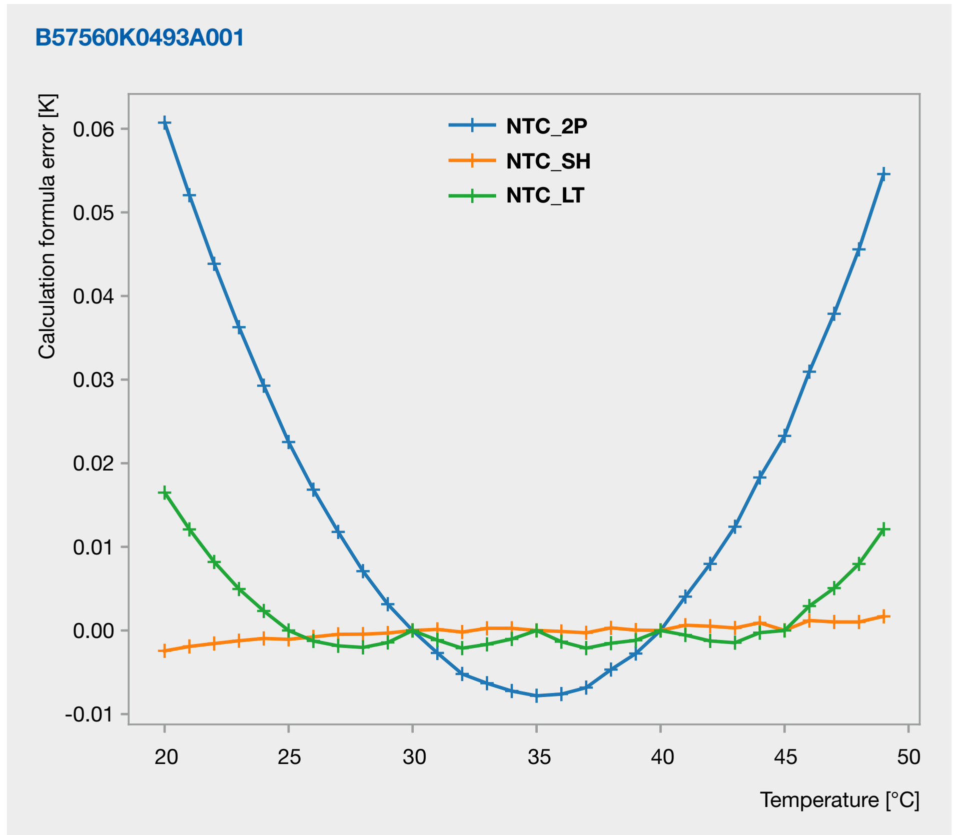


Figure 7: Difference between actual temperature and calculated temperature from the three different algorithms

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 = 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 we already have to deal with a temperature tolerance of:

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

For the algorithm in section 2.2 and 2.3 we used two and three data points to calculate the relevant coefficients for equation 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. But this is not the case: We will now see that with only one single measurement most of the manufacturing error 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 $\left(\frac{1}{T_R} - \frac{1}{T}\right)$:

$$\Delta_B T = \frac{1}{\alpha}\left|\Delta B \cdot \left(\frac{1}{T_R} - \frac{1}{T}\right)\right| = \frac{1}{4\%/\text{K}} \quad 2\% \cdot 3950\ \text{K} \cdot \left(\frac{1}{303{,}15\ \text{K}} - \frac{1}{306{,}15}\text{K}\right)\right| \leq \pm 0.12\ \text{K}$$

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 a factor 10 from 1.7 °C to 0.12 °C.

```
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

It is easy to extend the Python class NTC_2P() by two more functions to enable calibration. The code 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 python 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 case of the Steinhart-Hart approach in the Python 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 we made it available as part of the download package at the TDK NTC design tool pages.

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 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 my 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)
```

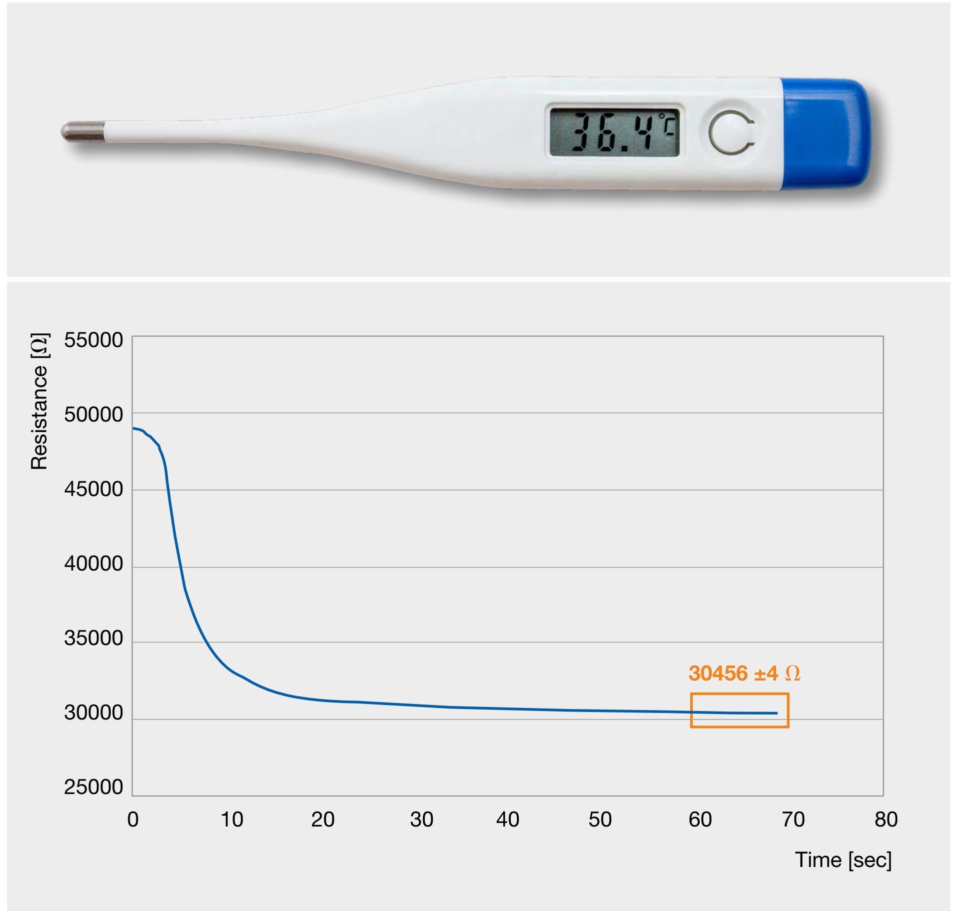


Figure 9: Example time plot for a calibration measurement

4. Conclusion

NTC Thermistor Sensors and probes can easily be integrated in digital circuits especially for remotesensing.

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

References

[1] **Texas Intruments.**
ADS111x Ultra-Small, Low-Power, I2C-Compatible, 860-SPS, 16-Bit ADCs
With Internal Reference, Oscillator, and Programmable Comparator
https://www.ti.com/product/ADS1115, 2018.

[2] **Adafruit Industries**
Adafruit 4 -Channel ADC Breakouts
https://learn.adafruit.com/adafruit-4-channel-adc-breakouts/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.

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

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

3.) NTC_LT.py: 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 our 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 digital converter. All class definitions include a class function for a single point software trimming of a given NTC.

Download the Python 3 classes tool here

DOWNLOAD

Address and contact info

Dr. Bernhard Ostrick
Produkt Development
Temperature & Pressure Sensors Business Group
bernhard.ostrick@tdk-electronics.tdk.com

www.tdk-electronics.tdk.com

Copyright © 2021 TDK Corporation. All rights reserved.
TDK logo is a trademark or registered trademark of TDK Corporation.

Publication date: 07/21

TDK