*Application Note*

# *How to Achieve ±1°C Accuracy or Better Across Temperature With Low-Cost TMP6x Linear Thermistors*

*Jesse Baker* *Temperature and Humidity Sensing*

**ABSTRACT**

Achieving high accuracy across temperature with traditional negative temperature coefficient (NTC) thermistors requires calibration at at least three different temperatures due to the non-linearity of NTCs across temperature. The TMP6x family of linear thermistors makes achieving high accuracy across temperature much more simple and cost effective. With the TMP6x thermistors, a single-point offset calibration can be performed at any temperature due to their linearity across temperature. Implementing oversampling and software filtering help eliminate noise from sensor readings and make it easier to see where the sensor truly is with respect to temperature. These methods can result in ±1°C accuracy or better across temperature with the TMP6x.

This document familiarizes the user with the steps needed to achieve high accuracy with the TMP6x thermistors. The document includes descriptions of a single-point offset correction, software methods of oversampling and filtering, as well as pseudocode for each to assist with evaluation and implementation.

## Table of Contents

## List of Figures

## Trademarks

All trademarks are the property of their respective owners.

## 1 Introduction

High accuracy and precision temperature measurements can be easily achieved at low cost with TMP6x linear thermistors using the steps outlined in this application note. As an example, the data in Figure 1-1 shows the average error across temperature for one unit after implementing the methods explained here. The steps to achieve high accuracy with the TMP6x are:

1. Use the 4th order polynomial from the TMP6x Thermistor Design Tool to convert the $V_{sense}$ voltage to temperature
2. Perform a single-point offset correction to increase accuracy; no temperature chamber or oil bath is needed.
3. Implement oversampling and software filtering to reduce noise and increase measurement precision. Use oversampling and software filtering together for best results.

Figure 1-1 shows the impact of a single-point offset, 32 × oversampling and software filtering with 0.1 alpha on accuracy and precision of one unit across temperature.

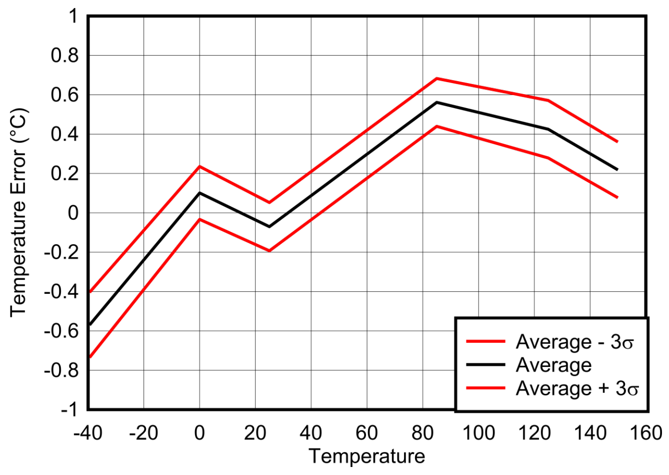

**Figure 1-1. Temperature Error ±3σ (Noise) Across Temperature**

The following sections provide brief descriptions of each individual step, as well as pseudocode to begin evaluating quickly.

## 2 Single-Point Offset Correction

The TMP6x thermistors are unique in that these thermistors have a fairly consistent error across temperature as compared to traditional negative temperature coefficient (NTC) thermistors, as shown in Figure 2-1. As a result, an offset correction can be performed at any temperature to remove tolerance errors from the TMP6x and other components in the thermistor biasing circuit, such as $V_{CC}$, $V_{Ref}$, ADC LSB, and $R_{Bias}$ errors. The errors remaining after the offset are the parts per million (ppm) errors of each component across temperature. Using low-ppm components in your system helps mitigate the impact of these remaining errors on temperature measurements.

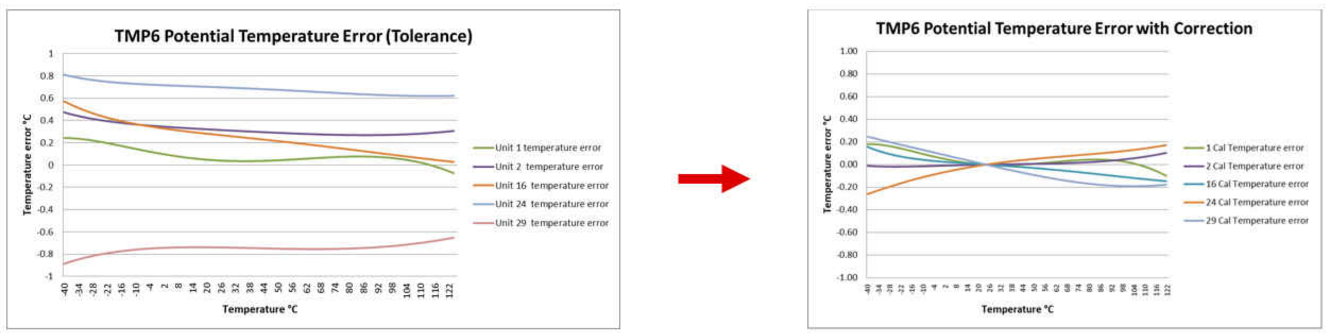

**Figure 2-1. TMP6x Error Before and After Single-Point Offset Correction**

To determine the offset for each TMP6x thermistor, a highly-accurate temperature reference is needed, such as the TMP117, an I2C temperature sensor with ±1°C max accuracy from –20°C to 50°C. Use software filtering during the offset calculation to determine a consistent and accurate offset. In the following pseudocode, 5,000 ADC samples of the voltage across the TMP6x are rapidly taken, and a software filter with an alpha of 0.001 is used to eliminate noise from the offset value. The final filtered voltage is converted to a temperature measurement using the 4th order polynomial from the Thermistor Design Tool. Finally, the offset is calculated as the difference between the TMP6x filtered temperature and the reference temperature, and that value is can be stored and applied to all future temperature measurements for that TMP6x thermistor.

```
int sensorPin = ; // EDIT indicate pin # for analog input to read VTEMP from TMP6
float Vbias = ; // EDIT indicate bias voltage
int ADC_bits = ; // EDIT indicate number of bits of resolution for ADC
int ADC_resolution = powf(2, ADC_bits) - 1; // number of ADC steps
int rawADC; // variable for measured ADC value
float VTEMP; // variable for measured voltage
float VTEMPfiltered; // variable for filtered voltage
float alpha = 0.001; // recommend using strong alpha value to get most robust temperature reading
int samples = 5000; // recommend at least 5,000 samples to give software filter time to stabilize
before calculating offset
float offset; // variable to store offset value
float TMP6filtered; // variable to hold the filtered temperature measurement
float referenceTemp; // temperature measurement from high-accuracy temperature reference (ex.
TMP117)

/* EDIT 4th order polynomial coefficients from Thermistor Design Tool */
float THRM_A0 = ;
float THRM_A1 = ;
float THRM_A2 = ;
float THRM_A3 = ;
float THRM_A4 = ;

/* in SETUP code */
rawADC = analogRead(sensorPin);
VTEMPfiltered = Vbias / ADC_resolution * rawADC; // initialize the VTEMPfiltered variable to the
first measured value to reduce ramp time

/* in MAIN code */
for(int i=0; i<samples; i++){
    rawADC = analogread(sensorPin);
    VTEMP = Vbias / ADC_resolution * rawADC;
    VTEMPfiltered = VTEMPfiltered - (alpha * (VTEMPfiltered - VTEMP)); // continually filter VTEMP
over 5,000 samples
}

TMP6filtered = (THRM_A4 * powf(VTEMPfiltered, 4)) + (THRM_A3 * powf(VTEMPfiltered, 3)) + (THRM_A2 *
powf(VTEMPfiltered, 2)) + THRM_A1 * VTEMPfiltered + THRM_A0

referenceTemp = ; // EDIT insert code to read reference temp here

offset = TMP6filtered - referenceTemp; // calculate offset as difference between TMP6filtered and
referenceTemp
```

## 3 Oversampling

Oversampling is one software method used to reduce the impact of noise. Simply put, oversampling indicates that rather than each individual sample resulting in a temperature measurement, N samples are accumulated and averaged, resulting in one averaged temperature measurement. This process reduces noise according to the number of oversamples, as shown by the difference between the raw and oversampled signals in Figure 3-1.

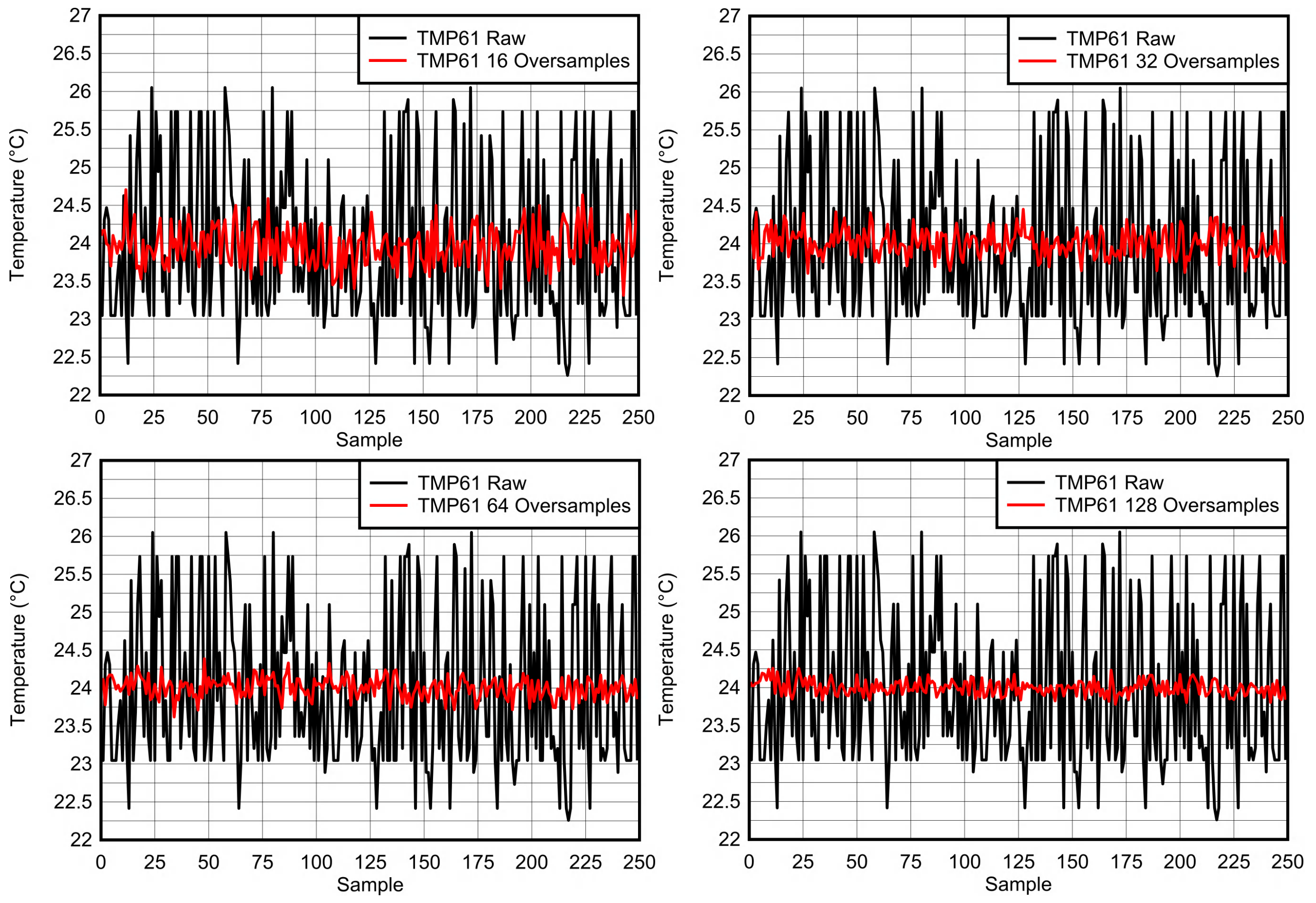

**Figure 3-1. Impact of 16 ×, 32 ×, 64 ×, and 128 × Oversampling on Noise**

The following pseudocode provides an example implementation of oversampling, which allows modification of the number of oversamples.

```
int sensorPin = ; // EDIT indicate pin # for analog input to read VTEMP from TMP6
float Vbias = ; // EDIT indicate Vref voltage for ADC
int ADC_bits = ; // EDIT number of bits of resolution for ADC
int ADC_resolution = pow(2, ADC_bits) - 1; // number of ADC steps

int rawADC; // variable for measured ADC value
float VTEMP; // variable for measured ADC voltage
float VTEMP_averaged; // variable for averaged ADC voltage
int oversamples = ; // EDIT number of oversamples
float TMP6oversampled; // variable for oversampled temperature reading

/* EDIT 4th order polynomial coefficients from Thermistor Design Tool */
float THRM_A0 = ;
float THRM_A1 = ;
float THRM_A2 = ;
float THRM_A3 = ;
float THRM_A4 = ;

for(int i = 0; i < oversamples; i++){
    rawADC = analogRead(sensorPin); // read ADC value
    VTEMP += Vbias / ADC_resolution * rawADC; // convert to voltage and sum for desired number of
oversamples
}

VTEMP_averaged = VTEMP / oversamples; // average the summed VTEMP
// convert voltage to temperature
TMP6oversampled = (THRM_A4 * powf(VTEMP_averaged, 4)) + (THRM_A3 * powf(VTEMP_averaged, 3)) +
(THRM_A2 * powf(VTEMP_averaged, 2)) + (THRM_A1 * powf(VTEMP_averaged, 1)) + THRM_A0;
VTEMP = 0; // reset VTEMP to 0 for next sample to be taken
```

## 4 Software Filtering

Filtering in software is also used for noise reduction. Software filtering acts as a low-pass filter and is implemented simply using the formula in Equation 1.

$$\text{filteredTemp} = \text{previousFilteredTemp} - (\alpha \times (\text{previousFilteredTemp} - \text{measuredTemp})) \tag{1}$$

With software filtering, the raw data is smoothed according to a smoothing factor, α. An α of 1 indicates no filtering, and an α of 0.99 allows most noise to be passed through, whereas an α of 0.01 eliminates a lot of noise. Figure 4-1 shows the impact of different levels of software filtering on noise reduction.

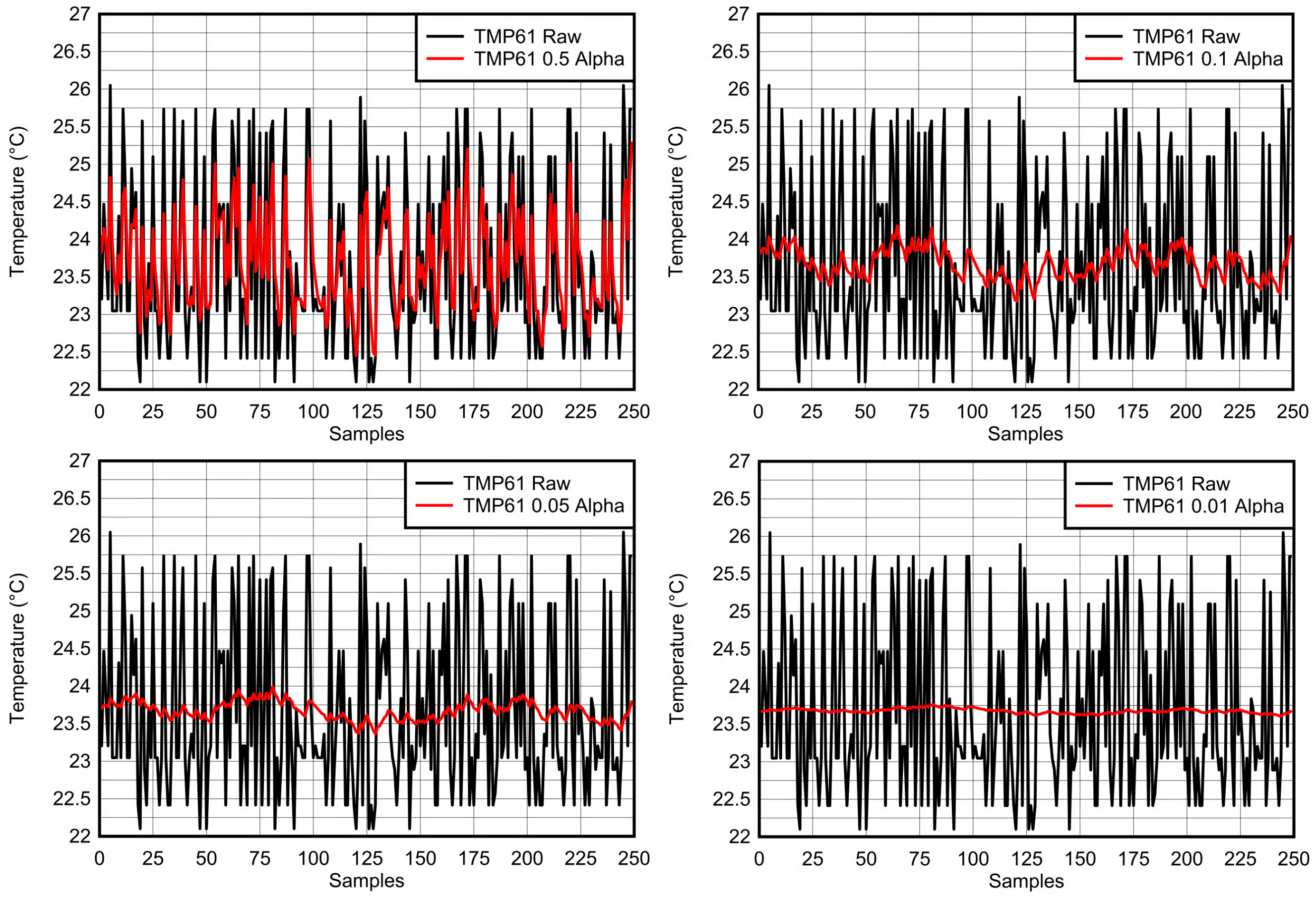

**Figure 4-1. Impact of Software Filters With Alpha Values 0.5, 0.1, 0.05, and 0.01 on Noise**

Using the following pseudocode, the α value of the filter can be easily adjusted to meet system requirements.

```
int sensorPin = ; // EDIT indicate pin # for analog input to read VTEMP from TMP6
float Vbias = ; // EDIT indicate Vref voltage for ADC
int ADC_bits = ; // EDIT number of bits of resolution for ADC
int ADC_resolution = pow(2, ADC_bits) - 1; // number of ADC steps

int rawADC; // variable for measured ADC value
float VTEMP; // variable for measured ADC voltage
float alpha = ; // EDIT 0.001 < alpha < 1, adjust as required. Smaller alpha means stronger filter
float TMP6raw; // variable for raw TMP6 temperature reading
float TMP6filtered; // variable for filtered temperature reading

/* EDIT 4th order polynomial coefficients from Thermistor Design Tool */
float THRM_A0 = ;
float THRM_A1 = ;
float THRM_A2 = ;
float THRM_A3 = ;
float THRM_A4 = ;

/* place in SETUP or INITIALIZATION code */
// this code initializes the filtered temperature to the initial raw temperature value
rawADC = analogRead(sensorPin);
VTEMP = Vbias / ADC_resolution * rawADC;
TMP6filtered = (THRM_A4 * powf(VTEMP, 4)) + (THRM_A3 * powf(VTEMP, 3)) + (THRM_A2 * powf(VTEMP, 2))
+ (THRM_A1 * powf(VTEMP, 1)) + THRM_A0;

/* MAIN code */
rawADC = analogRead(sensorPin); // read ADC value
VTEMP = Vbias / ADC_resolution * rawADC; // convert to voltage
// convert voltage to temperature
TMP6raw = (THRM_A4 * powf(VTEMP, 4)) + (THRM_A3 * powf(VTEMP, 3)) + (THRM_A2 * powf(VTEMP, 2)) +
(THRM_A1 * powf(VTEMP, 1)) + THRM_A0;
// calculate filtered temperature using previous filtered temp and raw temp
TMP6filtered = TMP6filtered - (alpha * (TMP6filtered - TMP6raw));
```

## 5 Noise Reduction With Oversampling and Software Filtering

Oversampling and software filtering can be implemented together to increase precision of temperature measurements. In Figure 5-1, each data point represents a different combination of oversampling and filtering, while the color of each point indicates the amount of noise reduction to compare the different combinations. Lighter dots indicate lower precision, whereas darker dots indicate a higher level of precision.

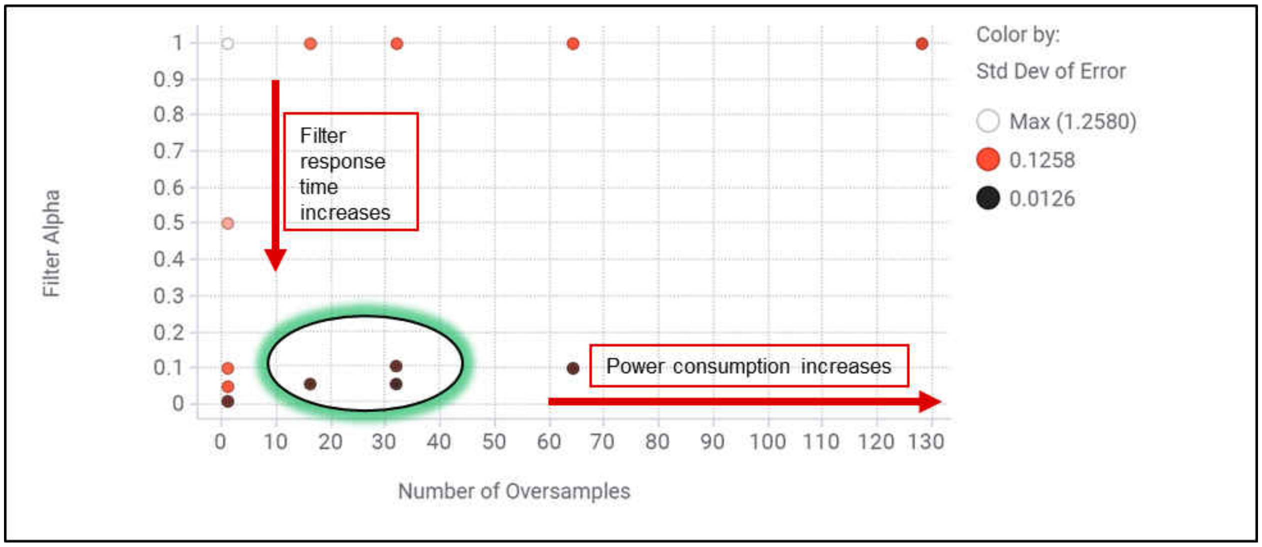

**Figure 5-1. Combined Impact of Oversampling and Software Filtering on Noise Reduction**

Start with 16 oversamples and a software filter with an α of 0.1 (inside the green oval in Figure 5-1), then adjust the number of oversamples and α value to meet your system needs. Use the following pseudocode to easily test different levels of oversampling and software filtering.

```
int sensorPin = ; // EDIT indicate pin # for analog input to read VTEMP from TMP6
float Vbias = ; // EDIT indicate Vref voltage for ADC
int ADC_bits = ; // EDIT number of bits of resolution for ADC
int ADC_resolution = pow(2, ADC_bits) - 1; // number of ADC steps

int rawADC; // variable for measured ADC value
float VTEMP; // variable for measured ADC voltagefloat VTEMP_averaged; // variable for averaged ADC
voltage
int oversamples = ; // EDIT number of oversamples
float alpha = ; // EDIT 0.001 < alpha < 1, adjust as required. Smaller alpha means stronger filter
float TMP6oversampled; // variable for oversampled TMP6 temperature reading
float TMP6oversampled_filtered; // variable for filtered temperature reading

/* EDIT 4th order polynomial coefficients from Thermistor Design Tool */
float THRM_A0 = ;
float THRM_A1 = ;
float THRM_A2 = ;
float THRM_A3 = ;
float THRM_A4 = ;

/* place in SETUP or INITIALIZATION code */
// this code initializes the oversampled_filtered temperature to the initial raw temperature value
rawADC = analogRead(sensorPin);
VTEMP = Vbias / ADC_resolution * rawADC;
TMP6oversampled_filtered = (THRM_A4 * powf(VTEMP, 4)) + (THRM_A3 * powf(VTEMP, 3)) + (THRM_A2 *
powf(VTEMP, 2)) + (THRM_A1 * powf(VTEMP, 1)) + THRM_A0;

/* MAIN code */
for(int i = 0; 1 < oversamples; i++){
    rawADC = analogRead(sensorPin); // read ADC value
    VTEMP += Vbias / ADC_resolution * rawADC; // convert to voltage and sum for desired number of
oversamples
}

VTEMP_averaged = VTEMP / oversamples; // average the summed VTEMP
// convert voltage to temperature
TMP6oversampled = (THRM_A4 * powf(VTEMP_averaged, 4)) + (THRM_A3 * powf(VTEMP_averaged, 3)) +
(THRM_A2 * powf(VTEMP_averaged, 2)) + (THRM_A1 * powf(VTEMP_averaged, 1)) + THRM_A0;
// calculate oversampled_filtered temperature using previous oversampled_filtered temp and
oversampled temp
TMP6oversampled_filtered = TMP6oversampled_filtered - (alpha * (TMP6oversampled_filtered -
TMP6oversampled));
VTEMP = 0; // reset VTEMP to 0 for next sample to be taken
```

## 6 Summary

The TMP6x family of thermistors enables high-accuracy measurements at low cost by utilizing a single-point calibration, oversampling, and software filtering. NTCs; however, require calibration at three or more different temperatures using a temperature chamber as a result of their non-linearity. Begin evaluating high-accuracy measurements with the TMP6x thermistors using the methods outlined in this application note.