ZSSC3240

Calibration Sequence and DLL

Introduction

The calibration DLL file described in this document is created to expedite the calibration process for the ZSSC3240. Section 2 gives a short overview for the main steps of calibration using the file. Section 3 covers how to implement a DLL (CalibrationL6.DLL) in customer-specific software.

Contents

List of Figures

List of Tables

2. Calibration Sequence

A typical calibration flow for the ZSSC3240 devices contains five steps in the following order:

1. Set-up and initialization
2. Data collection
3. Coefficient calculation
4. Memory programming
5. Verification

There are two approaches for data collection with the ZSSC3240:

- Using the raw measurement commands described in section 2.2.1 which requires a simpler initialization of the IC's memory (customer ID and AFE setup). This is the recommended approach.
- Using the IC-internal signal-correction math core. Thereby, the memory page must be utilized to feed the math core with proper initialization coefficients, and the IC-internal saturation mechanisms can significantly limit the dynamic range of the digital output.

See Figure 1 for a more detailed calibration flow graph.

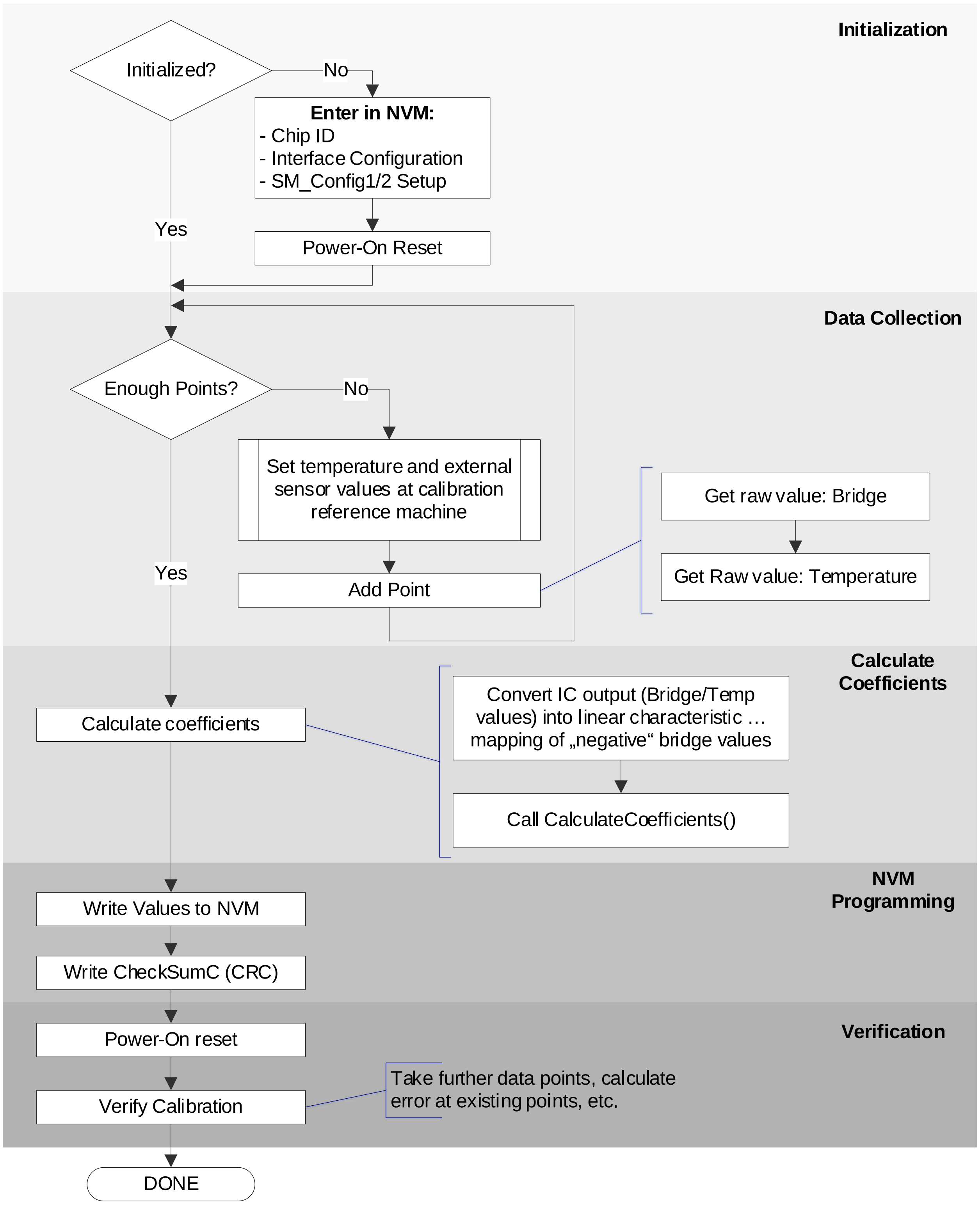
Initialization
Initialized?
No
Enter in NVM:
- Chip ID
- Interface Configuration
- SM_Config1/2 Setup
Yes
Power-On Reset
Data Collection
Enough Points?
No
Set temperature and external sensor values at calibration reference machine
Yes
Add Point
Get raw value: Bridge
Get Raw value: Temperature
Calculate Coefficients
Calculate coefficients
Convert IC output (Bridge/Temp values) into linear characteristic … mapping of „negative“ bridge values
Call CalculateCoefficients()
NVM Programming
Write Values to NVM
Write CheckSumC (CRC)
Verification
Power-On reset
Verify Calibration
Take further data points, calculate error at existing points, etc.
DONE

Figure 1. Calibration Flow Chart

2.1 Set-up and Initialization

2.1.1 Assigning a Unique Identification Number to the IC

This identification is programmed in ICs memory and can be used as an index in the database stored on the calibration PC. Such a database could contain all the raw values of external sensor readings (and temperature readings if applied or vice versa) for that part, as well as the according reference values for the calibration. See the *ZSSC3240 Datasheet*, for detailed description of the registers 0x00 (Cust_ID0) and 0x01 (Cust_ID1) dedicated to the customer for his product identification.

2.1.2 Analog Front End Configuration

Before useful raw data can be collected from the IC, the circuitry must be initialized. The initialization step involves setting the AFE (Analog Front End) configuration bits for the end application and optionally programming the math coefficients to their default value. See the *ZSSC3240 Datasheet* for detailed description for the single parameters of the AFE, and for the default settings of the AFE-parameters and coefficients, which have been already programmed during the wafer test.

2.1.3 Temperature Configuration

For a possible temperature measurement with the IC-internal temperature sensor, the default configuration is programmed into the temperature configuration registers. These default settings allow the full temperature range of -40°C to +125°C to be used.

2.2 Data Collection

The minimum number of calibration points used depends on the precision required and the behavior of the resistive bridge in use (it is normally between two and seven). There is no maximum number of calibration points that can be used; in general, taking more calibration points results in a better calibration.

Description of the standard set of calibration points are displayed in Figure 2.

- 2-point calibration is used to obtain only a gain and offset terms for bridge compensation with no temperature compensation for either term.
- 3-point calibration could be used either to
  - obtain the additional term SOT for 2nd order correction for the bridge (SOT_sens), but no temperature compensation of the bridge output.
  - temperature only is compensated, without using any external sensor
- 4-point calibration could be used to obtain bridge offset and gain, and both the Tco term and the Tcg term, which provides 1st order temperature compensation of the bridge offset and gain term. Additionally, the temperature sensor's offset and gain can be compensated based on the same calibration points.
- 5-point calibration could be used to obtain bridge sensor's gain, offset and 2nd-order term, Tco (bridge sensor related temperature offset term) and 2nd-order term that provides correction applied to the bridge's temperature coefficient's offset. Additionally, the temperature sensor's offset, gain and 2nd-order nonlinearity can be compensated based on the same calibration points.
- 6-point calibration could be used to obtain bridge sensor's gain, offset, Tcg, Tco, SOT_tco and SOT_tcg. Additionally, the temperature sensor's offset, gain and 2nd-order nonlinearity can be compensated based on the same calibration points.
- 7-point calibration could be used to obtain the complete set off supported signal correction coefficients for sensor bridge and IC-internal temperature sensor.

Table 1. Calibration Types

Type	Calculated Coefficients[a]	Required number of data points	
		Bridge	Temp
2 Points	OFFSET_S, GAIN_S	2	0
3 Points	OFFSET_S,GAIN_S, SOT_S	3	0
3 Points	OFFSET_T,GAIN_T, SOT_T	0	3
4 Points	OFFSET_S, GAIN_S, TCO, TCG, OFFSET_T, GAIN_T	2	2
5 Points	OFFSET_S, GAIN_S, TCO, OFFSET_T, GAIN_T, SOT_TCO, SOT_S, SOT_T	3	3
6 Points	OFFSET_S,GAIN_S, TCO, TCG, OFFSET_T, GAIN_T,SOT_TCO, SOT_TCG, SOT_T	2	3
7 Points	OFFSET_S,GAIN_S, TCO, TCG, OFFSET_T, GAIN_T,SOT_TCO, SOT_TCG,SOT_T, SOT_S	3	3

[a] Coefficients notation as used in the Calibration.dll / Calibration.h.

- *Gain_S*: External Sensor/Bridge gain term;
- *Offset_S:* External Sensor/Bridge offset term;
- *Tcg*: Temperature coefficient gain term;
- *Tco:* Temperature coefficient offset term;
- *SOT_tcg:* Second-order term for Tcg non-linearity;
- *SOT_tco:* Second-order term for Tco non-linearity;
- *SOT_sens:* Second-order term for bridge non-linearity;
- *Gain_T:* Gain coefficient for temperature;
- *Offset_T:* Offset coefficient for temperature;
- *SOT_T:* Second-order term for temperature source non-linearity.

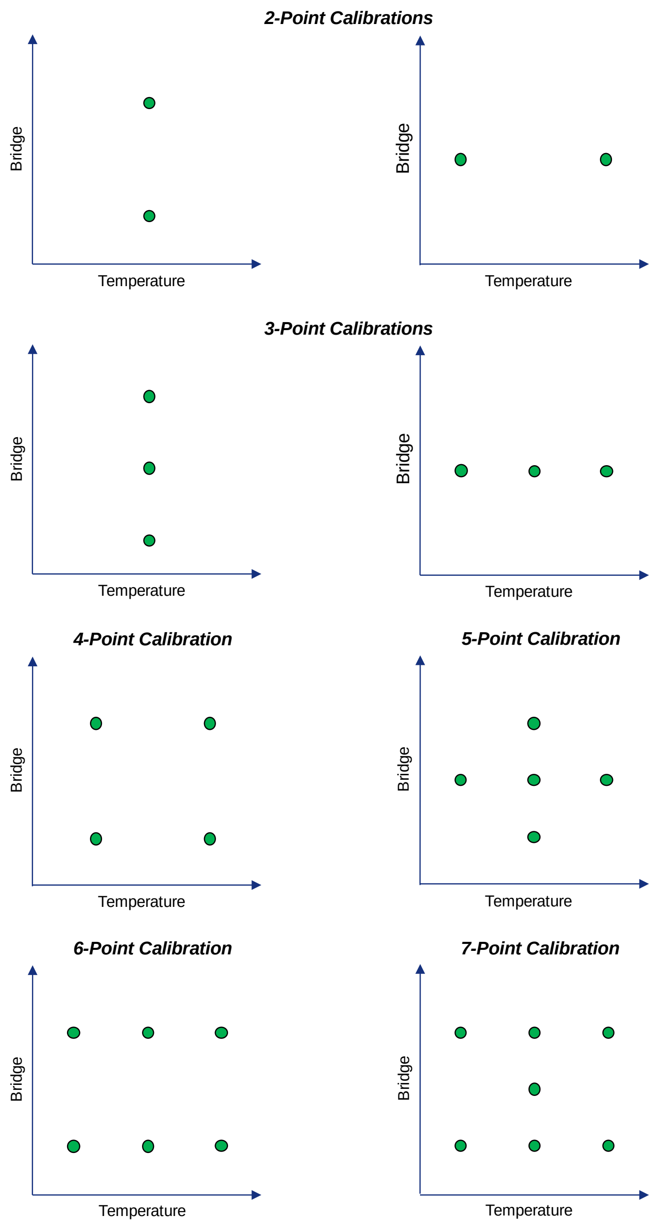

Figure 2. Calibration Point Locations for Selected Calibration Methods

Figure 2 shows the expected, recommended placement of calibration points for the different calibration options. The order of the points taken is not important; however, the number of points per temperature must be followed or the calibration might fail. It is important to keep the calibration points as orthogonal as possible to maximize calibration accuracy.

Further, the provided calibration DLL can also generate other subsets and combinations of calibration coefficients based on calibration points at different locations than described in Figure 2.

2.2.1 Data Collection by Raw Measurement Requests

The number of unique points (external sensor and/or temperature) at which calibration must be performed generally depends on the requirements of the application and the behavior of the resistive bridge in use. The minimum number of points required is equal to the number of bridge/temperature coefficients to be calculated. For a full calibration resulting in values for all seven possible bridge coefficients and three possible temperature coefficients, a minimum of seven pairs of bridge with temperature measurements must be collected.

2.2.1.1 Definition of Reference Values for Raw Measurements

The reference points for the resistive sensor calibration are usually defined in percent in relation to the full target application range. After that, they have to be converted into digital value relative to the full scale (FS) output of 24-bit, by a given function in the DLL.

The reference values for the raw temperature measurements are defined in degree Celsius (°C). In combination with user defined temperature limits (also in °C), the reference input for each point is then converted into the according digital reference value for the DLL.

For example, defining pressure reference points for calibration dependent on customers target range can be the following:

- Customer's target application range: 0 to 16bar
- Customer's pressure reference points: 2bar/6bar/14bar.
- Exact assignment would be:
  - 0bar -> 0% of the range
  - 16bar -> 100% of the range
- The defined reference points have the following assignment:
  - 2bar -> 12.5% of the range
  - 6bar -> 37.5% of the range
  - 14bar -> 87.5% of the range
- To add buffers for parasitic impact and to have integer percentage values for the calibration, it is recommended to change the points slightly as follows:
  - 2bar -> 15% of the range
  - 6bar -> 35% of the range
  - 14bar -> 85% of the range

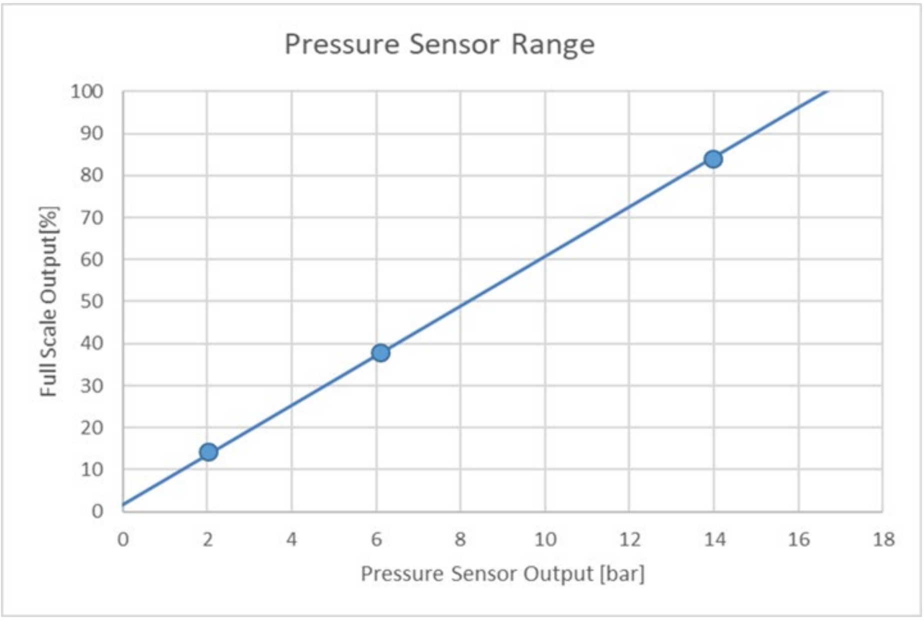

Figure 3. Assignment Input Resistive Range to SSC-output

To obtain the potentially best and most robust coefficients, it is recommended that measurement pairs (temperature vs. pressure) are collected near the outer corners of the intended operation range or at points which are located far from each other. It is essential to provide highly precise reference values as nominal, expected values. The measurement precision of the external calibration-measurement equipment must be ten times more accurate than the expected ZSSC3240 output precision after calibration in order to avoid precision losses caused by the nominal reference values (that is resistive sensor signal and temperature deviations).

Note: There is an inherent redundancy in the seven resistive sensor-related and three temperature-related coefficients. Since the temperature is a necessary output (which also needs correction), the temperature-related information is mathematically separated, which supports faster and more efficient DSP calculations during the normal usage of the sensor-IC system.

2.2.1.2 Raw Measurement Commands

Prior to the data collection, it is recommended to find the optimal AFE-configuration for the applied sensor and the target voltage input range, and then program it to the NVM configuration registers *SM_config1* and *SM_config2* (ZSSC3240). After AFE-configuration, raw data can be acquired. For it, the following two commands have to be used:

- for external sensor values:
  $A2_{HEX}$: Single raw data resistive sensor measurement for which the configuration is loaded from the *SM_config1* / *SM_config2* registers
- for temperature values:
  $A6_{HEX}$: Single raw data temperature measurement for which the configuration register is loaded from an internal temperature configuration register (preprogrammed by Renesas in NVM prior to IC delivery). If an external temperature sensor is configured, the configuration is loaded from the *extTemp_config1* / *extTemp_config2* registers.

2.2.1.3 Raw Data Output

The raw data measurement results are always MSB (Most Significant Bit)-aligned. The internal temperature sensor has a preconfigured setup with an ADC resolution of 13-bit. Figure 4 summarizes the recommended raw data process before passing it to the *CalculateCoefficients* function of the DLL.

In order to adapt both resistive and temperature raw values to the expected format (integer representation, 24-bit, MSB-aligned in the range of -2^23..2^23 in), they have to be converted from the two's complement representation to integer values in a range from -2^23..2^23.

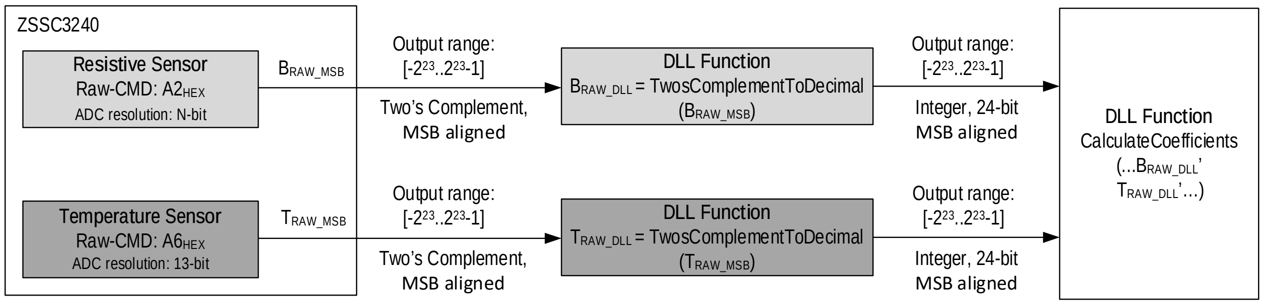
ZSSC3240
Resistive Sensor
Raw-CMD: A2HEX
ADC resolution: N-bit
Temperature Sensor
Raw-CMD: A6HEX
ADC resolution: 13-bit
B_RAW_MSB
T_RAW_MSB
Output range: [-2^23..2^23-1]
Two's Complement, MSB aligned
DLL Function
B_RAW_DLL = TwosComplementToDecimal (B_RAW_MSB)
T_RAW_DLL = TwosComplementToDecimal (T_RAW_MSB)
Output range: [-2^23..2^23-1]
Integer, 24-bit MSB aligned
DLL Function
CalculateCoefficients (…B_RAW_DLL', T_RAW_DLL'…)

Figure 4. Raw Data Handling for Coefficient Calculation (DLL)

2.3 Coefficient Calculations

The coefficients are calculated after all calibration data points are collected. The DLL exposes a C code interface and can be used directly from code (see section 3 for details). Features of the DLL are:

- Coefficient calculation
- Verification at calibration points
- Extended range verification

2.4 Programming NVM

After the coefficients have been calculated, they must be written to the NVM. The following table lists the commands necessary to program the coefficients to the according registers. Every coefficient is saved in the NVM in two different 16-bit registers, since each coefficient is a 24-bit wide value.

Table 2. Commands for Programming Coefficients and Final Settings of ZSSC3240

Command [Hex]	Data from Coefficients for the According Register	Description	Provided by
45	`coefficients[INDEX_OFFSET_S] & 0x00FFFF`	Offset_S[15:0]	DLL
46	`coefficients[INDEX_GAIN_S] & 0x00FFFF`	Gain_S[15:0]	DLL
47	`coefficients[INDEX_TCG] & 0x00FFFF`	Tcg[15:0]	DLL
48	`coefficients[INDEX_TCO] & 0x00FFFF`	Tco[15:0]	DLL
49	`coefficients[INDEX_SOT_TCO] & 0x00FFFF`	SOT_tco[15:0]	DLL
4A	`coefficients[INDEX_SOT_TCG] & 0x00FFFF`	SOT_tcg[15:0]	DLL
4B	`coefficients[INDEX_SOT_S] & 0x00FFFF`	SOT_sens[15:0]	DLL
4C	`coefficients[INDEX_OFFSET_T] & 0x00FFFF`	Offset_T[15:0]	DLL
4D	`coefficients[INDEX_GAIN_T] & 0x00FFFF`	Gain_T[15:0]	DLL
4E	`coefficients[INDEX_SOT_T] & 0x00FFFF`	SOT_T[15:0]	DLL
4F	`(coefficients[INDEX_OFFSET_S] & 0x7F0000) >> 8`	Offset_S[22:16]	DLL
	`(coefficients[INDEX_GAIN_S] & 0x7F0000) >> 16`	Gain_S[22:16]	DLL
	`(coefficients[INDEX_OFFSET_S] & 0x800000) ? 1 : 0`	Offset_S[23]	DLL
	`(coefficients[INDEX_GAIN_S] & 0x800000) ? 1 : 0`	Gain_S[23]	DLL

Data stream composition for MSB/SIGN register bits by the example of the offset and gain coefficients of the external sensor:

```
offset_s_msb    = (coefficients[INDEX_OFFSET_S] & 0x7F0000) >> 8;
gain_s_msb      = (coefficients[INDEX_GAIN_S] & 0x7F0000) >> 16;

if (coefficients[INDEX_OFFSET_S]<0) sign_offset_s        =  1;
else sign_offset_s =  0;

//the same if-else condition can be written as
// sign_offset_s = (coefficients[INDEX_OFFSET_S]<0) ? 1 : 0;
//this notation is used in the table below

if (coefficients[INDEX_GAIN_S]<0) sign_gain_s =  1;
else sign_gain_s =  0;

//define command and the register content
cmd = 0x4F;
//register data combination
data_0Fhex = sign_offset_s << 15 | offset_s_msb | sign_gain_s << 7| gain_s_msb;

//pseudo code for writing data to a specific (here 0x0D) register
//with the according command
write_mtp(cmd, data);
```

Numerical example:

```
// results from coefficients calculation
coefficients[INDEX_OFFSET_S] = -520831 // = 0x87F27F (24 bit sign-magnitude
                                    //representation)
coefficients[INDEX_GAIN_S] = 5880722        // = 0x59BB92 (24 bit sign-magnitude
                                    //representation)

offset_s_msb = 0x07
sign_offset_s = 1
gain_s_msb = 0x59
sign_gain_s = 0
data_0Fhex = 34649 = 0x8759
```

Note: The composition is equivalent for all further SIGN/MSB-registers.

Command [Hex]	Data from Coefficients for the According Register	Description	Provided by
50	(coefficients[INDEX_TCG] & 0x7F0000) >> 8	Tcg[22:16]	DLL
	(coefficients[INDEX_TCO] & 0x7F0000) >> 16	Tco[22:16]	DLL
	(coefficients[INDEX_TCG]<0) ? 1 : 0	Tcg[23]	DLL
	(coefficients[INDEX_TCO] <0) ? 1 : 0	Tco[23]	DLL
	data_10hex = register data combination as described in the example above in the example		
51	(coefficients[INDEX_SOT_TCO] & 0x7F0000) >> 8	SOT_tco[22:16]	DLL
	(coefficients[INDEX_SOT_TCG] & 0x7F0000) >> 16	SOT_tcg[22:16]	DLL
	(coefficients[INDEX_SOT_TCO] <0) ? 1 : 0	SOT_tco[23]	DLL
	(coefficients[INDEX_SOT_TCG] <0) ? 1 : 0	SOT_tcg[23]	DLL
	data_11hex = register data combination as described in the example above in the example		
52	(coefficients[INDEX_SOT_S] & 0x7F0000) >> 8	SOT_sense[22:16]	DLL
	(coefficients[INDEX_OFFSET_T] & 0x7F0000) >> 16	Offset_T[22:16]	DLL
	(coefficients[INDEX_SOT_S] <0) ? 1 : 0	SOT_sens[23]	DLL
	(coefficients[INDEX_OFFSET_T] <0) ? 1 : 0	Offset_T[23]	DLL
	data_12hex = register data combination as described in the example above in the example		
53	(coefficients[INDEX_GAIN_T] & 0x7F0000) >> 8	Gain_T[22:16]	DLL
	(coefficients[INDEX_SOT_T] & 0x7F0000) >> 16	SOT_T[22:16]	DLL
	(coefficients[INDEX_GAIN_T] <0) ? 1 : 0	Gain_T[23]	DLL
	(coefficients[INDEX_SOT_T] <0) ? 1 : 0	SOT_T[23]	DLL
	data_13hex = register data combination as described in the example above in the example		

2.5 Verification

The DLL interface provides verification at calibration time (see section 0). To verify if results are consistent with expected results, also perform an online verification at a different bridge measurand / temperature combination than was used for calibration.

3. CalibrationL6.DLL

The CalibrationL6.DLL's properties, interfacing and variable declaration, and the available routines with the respective returns of the available methods are characterized in detail. The main focus in this document is to enable the reader to integrate the DLL in a customer software environment for production purposes.

3.1 DLL Setup

Take the following setup steps to use the CalibrationL6.DLL in a user program:

1. Declare all functions to be used from the DLL:
   a. In C/C++, link *CalibrationL6.lib* into the final executable.
   b. In VB (Visual Basic), add *CalibrationL6.DLL* as a reference and verify that it is in the path.
2. Create *CalibrationL6.h* that must contain the same declarations for the functions used in *CalibrationL6.DLL*. The user's program must be setup to use Windows™ calling conventions (stdcall), not "C" style calling conventions (cdecl).

All functions listed in section 3 can be called as if they were local functions.

3.2 DLL Use

CalibrationL6.DLL typically is used for the following calibration steps:

1. Data Conversion: all raw and target data input for both bridge and temperature (if applicable) must be converted into the correct format, see section 0.
2. Coefficient Calculation: The converted data along with control information is passed to the *CalculateCoefficients* method which generates all necessary coefficients, see section 3.3.3.
3. Verification: The coefficients are verified both for accuracy and proper operation across the entire region of operation. The *CalibrationL6.DLL* provides methods to do this verification offline, see section 0.

3.2.1 Using Customer Default Values as Coefficients

The *CalibrationL6.DLL* library supports calibration using customer-calculated default values; these values can be applied to all calibrations without recalculating each time allowing one less calibration point for every used default value. The pre-condition for using customer default values is a known, repeatable sensor characteristic. The result of a calibration using default values is always less accurate than a complete calibration. To use a default value during calibration, do not select coefficient for calculation.

3.3 CalibrationL6.DLL Application Programming Interface (API)

3.3.1 Constants used with CalibrationL6.DLL

Within CalibrationL6.DLL many different enumerations are used to clarify the control and separation of data going to and from the DLL.

3.3.1.1 COEFFICIENT_COUNT

COEFFICIENT_COUNT is a constant that represents the number of coefficients. All coefficient arrays passed to CalibrationL6.DLL are expected to be of size COEFFICIENT_COUNT.

Example: Declaration of an array of integers for the coefficients and initialize the array to 0.

```
int coefficients[COEFFICIENT_COUNT] = {0}; //c compiler will 0 fill remaining entries
```

3.3.1.2 Calibration Type

The programmable coefficients have the listed flag values (see the following C code declaration) in the DLL. The most common combinations of coefficients are shown in the source code *Example* of this section. The type of calibration desired is indicated through the coefficients selected for calibration. For best results, use the pre-defined combinations. The coefficients can be individually OR’ed together in order to form other calibration types.

C code declaration:

```
#define CO_OFFSET_S                     0x1
#define CO_GAIN_S                       0x2
#define CO_TCG                          0x4
#define CO_TCO                          0x8
#define CO_SOT_TCO                      0x10
#define CO_SOT_TCG                      0x20
#define CO_SOT_S                        0x40
#define CO_OFFSET_T                     0x80
#define CO_GAIN_T                       0x100
#define CO_SOT_T                        0x200
```

Example: The following C code lines show applicable combinations of coefficients and a possible definition of a variable which passes this information validly to the *CalculateCoefficients* method.

```
int errorcode;
int negCoeffs;

// Variable definition for required coefficients
int P2_S = (CO_OFFSET_S|CO_GAIN_S);
int P3_S = (CO_OFFSET_S|CO_GAIN_S|CO_SOT_S);
int P3_T = (CO_OFFSET_T|CO_GAIN_T|CO_SOT_T);
int P4_S = (CO_OFFSET_S|CO_GAIN_S|CO_TCO|CO_TCG|CO_OFFSET_T|CO_GAIN_T);
int P5_S = (CO_OFFSET_S|CO_GAIN_S|CO_TCO|CO_OFFSET_T|CO_GAIN_T|CO_SOT_TCO|CO_SOT_S|CO_SOT_T);
int P6_S =
(CO_OFFSET_S|CO_GAIN_S|CO_TCO|CO_TCG|CO_OFFSET_T|CO_GAIN_T|CO_SOT_TCO|CO_SOT_TCG|CO_SOT_T);
int P7_S =
(CO_OFFSET_S|CO_GAIN_S|CO_TCO|CO_TCG|CO_OFFSET_T|CO_GAIN_T|CO_SOT_TCO|CO_SOT_TCG|CO_SOT_T|CO_SOT_S);
…

// calculate just bridge coefficients -> P3_S
// possible function call
```

```
errorcode = CalculateCoefficients(      coefficients,
                                        &negCoeffs
                                        2,
                                        P3_S,
                                        0,
                                        rawBridge,
                                        desiredBridge,
                                        rawDummy,
                                        desiredDummy, /* Not calibrating anything with temp */
                                        );
```

3.3.1.3 Indexes for Coefficients

After calculating coefficients, the CalibrationL6.DLL provides them in a certain order in the coefficients array. The access with these indexes returns the signed value of each coefficient.

C code declaration:

```
//INDEXES for coefficients array
#define INDEX_OFFSET_S          0
#define INDEX_GAIN_S            1
#define INDEX_TCG                       2
#define INDEX_TCO                       3
#define INDEX_SOT_TCO           4
#define INDEX_SOT_TCG           5
#define INDEX_SOT_S             6
#define INDEX_OFFSET_T          7
#define INDEX_GAIN_T            8
#define INDEX_SOT_T             9
```

Example: Accessing the OFFSET_S coefficient value after calculation with *CalculateCoefficients* method:

```
//assuming int coefficients[COEFFICIENT_COUNT]; has been previously declared
int offset_s = coefficients[INDEX_OFFSET_S];
```

3.3.1.4 Sign Flags of the Coefficients

The sign flags allow excluding a certain sign from the representative ‘sign number’, which contains the sign information for all coefficients. The coefficients themselves are signed, too. This ‘sign number’ makes data processing more comfortable. Gain coefficients do not have a flag for negative presentation, the results are always positive.

C code declaration:

```
//FLAGS for negCoeffs
#define NEG_SOT_S               0x1
#define NEG_SOT_TCO             0x2
#define NEG_SOT_TCG             0x4
#define NEG_SOT_T               0x8
#define NEG_TCO                 0x10
#define NEG_TCG                 0x20
#define NEG_OFFSET_S            0x40
#define NEG_OFFSET_T            0x80
```

Example:

```
int negSOT_S =0;

//negSOT_S=0 when the coefficient is positive, = 1 when it’s negative.
negSOT_S = negCoeffs & NEG_SOT_S;
```

3.3.2 Conversion Routines

The following conversion routines are used for translation of an input value into the necessary format to complete the calculations.

3.3.2.1 Bridge Conversion Routines

Table 3. Overview of the Routines

Name	Description
ConvertBridgeFromPercent	Converts a percentage value [0,100] into the proper domain for use by *CalibrationL6.DLL.* 100 percent correspond to the full scale output (16777215 = 2^24-1) of the 24-bit wide IC output
ConvertBridgeToPercent	Converts result from the IC (corrected measurement) or DLL's calculation domain into a percentage reading for use in error calculations.

The percentage declarations for the bridge input are useful for defining the common range of the measured item, for example, pressure. For calculation or verification routines listed in sections 3.3.3 and 0, the sensor inputs must be processed through *ConvertBridgeFromPercent* routine which maps the bridge sensor precentral values (0% to 100%) to the full scale range of 24-bit.

C code declaration:

```
double  ConvertBridgeFromPercent(double percent);
```

Returns: The desired (reference) sensor value in counts according to the input in percent.

Example: One calibration input represents the desired and reference value of 10%. To convert this sensor value for valid use in further process of coefficients calculation, this function has to be applied:

```
double desired_s1 = ConvertBridgeFromPercent( 10.0);
```

ConvertBridgeToPercent can be used to convert any output from *CalibrationL6.DLL* back into the percentage domain for error analysis. This routine should be used for the external sensor output after calibration. Otherwise the percentage numbers is meaningless.

C code declaration:

```
double ConvertBridgeToPercent(double codes);
```

Returns: The sensor value in percent according to the input in code is provided.

Table 4. Parameter Bridge Routines

Parameter	Description
codes	24-bit digital result value from the IC or DLL's calculation (corrected measurement).
percent	Bridge value in percent, referring to the applied measurement range.

3.3.2.2 Temperature Conversion Routines

Table 5. Overview of the Routines

Name	Description
ConvertTempFromDegrees	Converts a Celsius value [-45,150] into the proper domain for use by CalibrationL6.DLL. User entered limit for the maximum temperature corresponds to the full scale output (16777215 = 2^24-1) of the 24-bit wide IC output.
ConvertTempToDegrees	Converts result from the IC (corrected measurement) or DLLs domain back into Celsius to use in error calculations or to display values in Celsius.

All '°C' temperature inputs must be run through the *ConvertTempFromDegrees* function before coefficients calculation. It expects a value between [-45, +150°C]. The result in code is saved to the variable, which is passed on first place as a reference.

C code declaration:

```
__int32 ConvertTempFromDegrees(  double *tempInCodes,
                                double tempInDegrees,
                                double minTemp,
                                double maxTemp);
```

Returns: an error code denoting the status of the calculations. '0' is returned if the method was passed successfully. '1' is returned if the input parameters are out of the expected ranges.

Example: During calibration, an environmental temperature of 50°C is applied as a calibration point. It needs to be converted for further coefficient determination. The limits for minimum and maximum temperature have to be provided to the function.

```
double desiredTemp;
int errorcode = 0;

errorcode = ConvertTempFromDegrees(&desiredTemp, 50.0, -40.0, 125.0);
```

ConvertTempToDegrees can be used to convert a 24-bit temperature as returned by *GetCorrectedTemp* into degrees Celsius.

C code declaration:

```
__int32 ConvertTempToDegrees(    double *tempInDegrees,
                                 __int32 tempInCodes,
                                 double minTemp,
                                 double maxTemp);
```

Returns: an error code denoting the status of the calculations. '0' is returned if the method was passed successfully. '1' is returned, if the input parameters are out of expected ranges.

Example: It is assumed that calibration is performed successfully. The coefficients are calculated and stored in coefficients [COEFFICIENT_COUNT].

```
double tempCorrectedCodes;
double tempDegreesC;
int errorcode = 0;

tempCorrectedCodes = GetCorrectedTemp(coefficients, 320000);
errorcode += ConvertTempToDegrees(&tempDegreesC, tempCorrectedCodes, -40, 85);
```

Table 6. Parameter Temperature Routines

Parameter	Description
*tempInCodes	Pointer to the variable where the calculated raw temperature value is stored.
tempInDegrees	Temperature in Celsius to be converted to codes.
minTemp	The lower temperature limit of the calibration range, in Celsius.
maxTemp	The upper temperature limit for of the calibration range, in Celsius.

3.3.2.3 Raw Values Conversion

Table 7. Overview of the Routine

Name	Description
TwosComplementToDecimal	Converts a raw measurement value into a signed integer number in the range [-2^23..2^23-1].

Raw bridge measurement results are provided from the ZSSC3240 as N-bit two's complement numbers, where N is the customer configured ADC-resolution. For a proper input to the *CalculateCoefficients* function or for common display in as a signed integer values, they have to be converted accordingly. Further details are described in section 0.

For the conversion from a 24-bit two's complement value to a 24-bit decimal value, the *TwosComplementToDecimal* function can be used.

C code declaration:

```
__int32 TwosComplementToDecimal (__int32 input);
```

Returns: Digital value in signed magnitude representation.

Example:

```
__int32 testTwosComp = 0;
__int32 signMagn = 0;

testTwosComp = 0xfffff6;
signMagn = TwosComplementToDecimal(testTwosComp);
// signMagn = -10

testTwosComp = 0x7000A3;
signMagn = TwosComplementToDecimal(testTwosComp);
// signMagn = 7340195

testTwosComp = 0x5;
signMagn = TwosComplementToDecimal(testTwosComp);
// signMagn = 5

testTwosComp = 0x800005;
signMagn = TwosComplementToDecimal(testTwosComp);
// signMagn = -8388603
```

3.3.3 Coefficients Calculation

CalculateCoefficients is the main function for doing the actual calibration calculations. It determines a set of coefficients that provides calibrated output based on the provided set of data points. This function provides the calibrated coefficients, which can be used in all of the verification methods listed in section 0.

C code declaration:

```
__int32 CalculateCoefficients(__int32 coefficients[COEFFICIENT_COUNT],
                                          __int32 *negCoeffs
                                          __int32 numPoints,
                                          __int32 selCoeffs,
                                          __int32 calType,
                                          double *bridgeRaw,
                                          double *bridgeDesired,
                                          double *tempRaw,
                                          double *tempDesired);
```

Returns: an error code denoting the status of the calculations. ‘0’ is passed if the method was passed completely.

Before using the *CalculateCoefficients* function, the collected raw data must be converted to the expected format. For further details on the IC-provided measurement data, see section 2.2.

Example:

```
int errorcode = 0;

int numPoints = 2;
int negCoeffs=0;

double rawBridge[2], desiredBridge[2];

// temperature input not relevant
double rawDummy[2] = = {NULL,NULL};
double desiredDummy[2] = {NULL,NULL};


int selCoeffs = CO_OFFSET_S | CO_GAIN_S;

// set coefficient array to zero
int coefficients[COEFFICIENT_COUNT] = {0};
```

```
// calibration type, default value
int calType = 0;

// raw data as double values
rawBridge[0] = -10000.0;
rawBridge[1] = 8236410.0;

// convert percentage reference values into the digital representative
desiredBridge [0] = ConvertBridgeFromPercent(10.0);
desiredBridge [1] = ConvertBridgeFromPercent(90.0);

// run coefficients calculation
errorcode = CalculateCoefficients(       coefficients,
                                         &negCoeffs,
                                         numPoints,
                                         selCoeffs,
                                         calType,
                                         rawBridge,
                                         desiredBridge,
                                         rawDummy,        /* Not calibrating anything with temp */
                                         desiredDummy     /* Not calibrating anything with temp */
                                                                 );
/************resulting coefficients******
coefficients[0] = coefficients[INDEX_OFFSET_S] = -1028301
coefficients[1] = coefficients[INDEX_GAIN_S] = 3413303
errorcode = 0
*****************************************/
```

Table 8. Parameter CalculateCoefficients Function

Parameter	Description
coefficients[COEFFICIENT_COUNT]	This array contains the calculated coefficients (functions' return). The array must be zero-filled prior to calling CalculateCoefficients unless using default values.
*negCoeffs	Pointer to the representative sign parameter, with bitwise negative coefficient flags.
numPoints	Number of calibration points used.
selCoeffs	In binary representation, this parameter indicates which coefficient is to be calculated.
calType	The type of calibration desired. A default value of 0 is recommended, which represents the parabolic correction function and provides the best calculation approach.
*bridgeRaw [a]	Array of raw sensor values. Must be converted for DLL input and have the length of numPoints. If not calibrating for bridge correction, the array elements can be NULL.
*bridgeDesired [a]	Array of target sensor values. Must be converted for DLL input and have the length of numPoints If not calibrating for bridge correction, the array elements can be NULL.
*tempRaw [a]	Array of raw temperature values. Must be converted for DLL input and have the length of numPoints. If not calibrating for temperature correction, the array elements can be NULL.
*tempDesired [a]	Array of target temperature values. Must be converted for DLL input and have the length of numPoints. If not calibrating for temperature correction, the array elements can be NULL.

[a] The array must have matching indices to the according calibration points.

3.3.4 Verification Routine

The function checks whether the DLL calculation produced coefficients, or has a size exceeding the destined dimensions. It is recommended to apply this function after each calculation of coefficients.

C code declaration:

```
__int32 VerifyCoefficients(const __int32 coefficients[COEFFICIENT_COUNT]);
```

Returns: An __int32 error code denoting the status of the calculations: '1' on failure, '0' on success.

Example:

```
int errorcode = 0;
errorcode = VerifyCoefficients(coefficients);

if (errorcode != 0) // coefficients out of range
```

3.3.4.1 GetCorrectedTemp

GetCorrectedTemp calculates the calibrated temperature output based on the given calculated coefficients and a raw temperature value.

C code declaration:

```
double GetCorrectedTemp(const __int32 coefficients[COEFFICIENT_COUNT], double rawTemp);
```

Returns: The calibrated temperature in double-precision floating-point format is provided. It can be converted to Celsius using the *ConvertTempToDegree* function, see section 0.

3.3.4.2 GetCorrectedBridge

GetCorrectedBridge calculates the calibrated bridge output based on the given calculated coefficients and raw sensor and raw temperature values.

C code declaration:

```
double GetCorrectedBridge(const __int32 coefficients[COEFFICIENT_COUNT],
                              double rawBridge, double rawTemp);
```

Returns: The calibrated output in double-precision floating-point format is provided. It can be converted to percentage using Bridge Conversion Routines, see section 3.3.2.1.

Example: Assuming a seven point bridge/temperature calibration has been accomplished with raw data (rawBridge[], rawTemp[]) and the result of a set of valid coefficients. Then a possible verification of the target accuracy (here: 1.5% for the external bridge sensor and 3°C for temperature) at the calibration points could be done as the below source code shows. Such verification does not include the inaccuracies caused by the sensor and measurement, but the deviations caused by correction calculation.

```
// rawBridge[], rawTemp[] -> contain raw bridge/temperature data
// coefficients[] -> contain a set of valid coefficients
// refTempDeg[] -> contain reference temperature values in degree Celsius
// rawBridgePerc[] -> contain reference pressure values in percent

int errorcode = 0;

double outBridgeCodes, outBridgePerc, outTempCodes, outTempDeg;

// loop over calibration points
for(int i=0; i<3; i++) {

        //Verify Temperature accuracy
        outTempCodes = GetCorrectedTemp(coefficients, rawTemp[i]);
        errorcode += ConvertTempToDegrees(&outTempDeg, outTempCodes, -40.0, 125.0);

        // check ambient temperature accuracy comparing degC values
        // between measured and reference values
        if( fabs(refTempDeg[i]-outTempDeg) > 3.0 ) //ERROR

        outBridgeCodes = GetCorrectedBridge(coefficients, rawBridge[i], rawTemp[i]);
        outBridgePerc = ConvertBridgeToPercent(outBridgeCodes);
        // check external sensor accuracy comparing percentage values
        // between measured and reference values
        if( fabs(outBridgePerc-rawBridgePerc[i]) > 1.5 ){…} //ERROR

};
```

3.3.4.3 BackCalcRawTemp

BackCalcRawTemp is the inverse function of GetCorrectedTemp. It calculates the raw temperature value based on the given calculated coefficients and a corrected temperature value.

C code declaration:

```
__int32 BackCalcRawTemp(const __int32 coefficents[COEFFICIENT_COUNT],
                        double  *rawTemp, double correctedTempInDeg,
                        double minTemp, double maxTemp);
```

Returns: an error code denoting the status of the calculations. ‘0’ is returned if the method was passed successfully.‘1’ is returned, if the input parameters are out of expected ranges.

3.3.4.4 BackCalcRawBridge

BackCalcRawBridge is the inverse function of GetCorrectedBridge. It calculates the raw bridge value based on the given calculated coefficients and a corrected temperature value. Since the correction of bridge values is processing also raw temperature values for specific calibration types, BackCalcRawBridge expects also the passing of it.

C code declaration:

```
__int32 BackCalcRawBridge(const __int32 coefficents[COEFFICIENT_COUNT],
                                  double  * rawBridge,
                                  double correctedBridgeInPerc,
                                  double rawTemp);
```

Returns: an error code denoting the status of the calculations. ‘0’ is returned if the method was passed successfully. ‘1’ is returned, if the input parameters are out of expected ranges.

Example:

```
// rawBridge[], rawTemp[] -> contain raw bridge/temperature data
// coefficients[] -> contain a set of valid coefficients
// caliPoints -> number of calibration points
// T_min,T_max -> temperature calibration limits

double correctedTempInCodes[caliPoints], correctedTempInDegC[caliPoints];
double correctedBridgeInCodes[caliPoints], correctedBridgeInPerc[caliPoints];
double rawT = 0, rawB = 0;

// correction functions applied in this loop calculating corrected output
for (i = 0;i<caliPoints;i++){

        correctedTempInCodes[i] = GetCorrectedTemp(coefficients, rawTemp[i]);
        // convert corrected codes into degree celsius
        ConvertTempToDegrees(&correctedTempInDegC[i], (int)correctedTempInCodes[i], T_min, T_max);

        correctedBridgeInCodes[i] = GetCorrectedBridge (coefficients, rawBridge[i] , rawTemp[i]);
        // convert corrected codes into percent
        correctedBridgeInPerc[i] = ConvertBridgeToPercent(correctedBridgeInCodes[i]);
}

// back calculation functions applied in this loop calculating raw values
// from corrected degree celsius/percentage values
for (i = 0;i<cali_points;i++){

        BackCalcRawTemp(coefficients, &rawT, CorrectedTempInDegC[i],  T_min, T_max );
        BackCalcRawBridge(coefficients, &rawB, correctedBridgeInPerc[i], rawT);

        // origin and recalculated raw values should be the same
        // rawTemp[i] == rawT -> True
```

```
                // rawBridge[i] == rawB -> True
        }
```

Table 9. Parameter BackClacRawTemp/BackCalcRawBridge Functions

Parameter	Description
coefficients[COEFFICIENT_COUNT]	This array contains the applied coefficients.
*rawTemp [a]	Array of raw temperature values (functions' return).
correctedTempInDeg	The corrected temperature measurement output, should be provided in degree Celsius
*rawBridge [a]	Array of raw sensor values. Must be converted for DLL input and have the length of numPoints. If not calibrating for bridge correction, the array elements can be NULL.
correctedBridgeInPerc	The corrected bridge measurement output, should be provided in percent
minTemp	The lower temperature limit of the calibration range, in Celsius.
maxTemp	The upper temperature limit for of the calibration range, in Celsius.
[b] The array must have matching indices to the according calibration points.	

4. Glossary

Term	Description
AFE	Analog Front End
API	Application Programming Interface
CMD	Command
CRC	Cyclic Redundancy Check
DLL	Dynamic-Link Library: an executable file that enables programs to share code and resources for completing specific tasks
FS	Full Scale
GUI	Graphical User Interface
IC	Integrated Circuit
ID	Identifier
LSB	Least Significant Bit
MSB	Most Significant Bit
NVM	Non Volatile Memory
PC	Personal Computer
SSC	Sensor Signal Conditioner
T	Temperature
VB	Visual Basic

5. Revision History

Rev.	Date	Description	
		Page	Summary
1.0	Apr.15.20		Initial release

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/

Trademarks