2.1.3 Temperature Configuration

For temperature measurements, the default configuration is programmed in the wafer test in the temperature configuration register, which is hidden for customers. These default settings provide full -40 +125°C temperature range.

2.2 Data Collection

The minimum number of calibration points required varies between two and usually seven. This depends on the precision required and the behavior of the capacitive sensor in use. 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 (see also Figure 2 and the *ZSSC3230 Datasheet*):

- 2-points calibration: obtains a gain and offset terms for sensor compensation with no temperature compensation for either term.
- 3-points calibration:
  - Obtains the additional term SOT for second order correction for the sensor (SOT_sensor), but no temperature compensation of the sensor output.
  - Temperature is compensated, without using any external sensor.
- 4-points calibration: obtains sensor offset and gain, and both the Tco term and the Tcg term, which provides first order temperature compensation of the sensor offset and gain term. Additionally, the temperature sensor's offset and gain can be compensated based on the same calibration points.
- 5-points calibration: obtains sensor's gain, offset and second order term, Tco (capacitive sensor related temperature offset term) and second order term that provides correction applied to the sensor's temperature coefficient's offset. Additionally, the temperature sensor's offset, gain and second order nonlinearity can be compensated based on the same calibration points.
- 6-points calibration: obtains capacitive sensor's gain, offset, Tcg, Tco, SOT_tco and SOT_tcg. Additionally, the temperature sensor's offset, gain and second order nonlinearity can be compensated based on the same calibration points.
- 7-points calibration: obtains the complete set off supported signal correction coefficients for the capacitive sensor and IC-internal temperature sensor.

Table 1. Calibration Types

Type	Calculated Coefficients[a]	Required number of data points	
		Cap Sensor	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.

The following list explains the meaning of all possible coefficients:

- Gain_S: External Sensor gain term;
- Offset_S: External Sensor 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_sensor: Second-order term for sensor 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 ZSSC3230

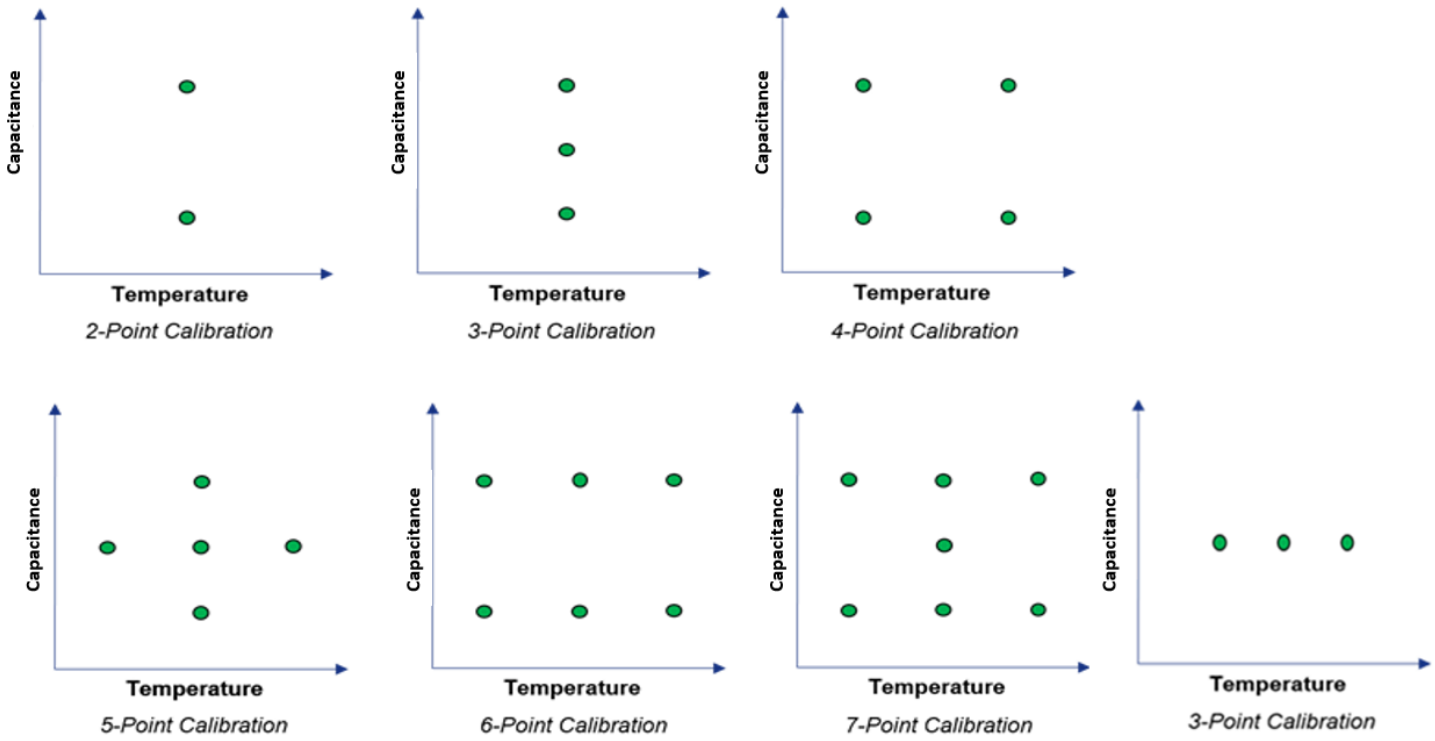


Figure 2 shows the expected, recommended placement of calibration points for the different calibration options. The order of the points taken is not important; but the number of points per temperature must be followed to have a relevant calibration. Important note: keep the calibration points as orthogonal as possible to maximize calibration accuracy.

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

Each measurement point is the pair of a capacitive raw measurement and a temperature value.

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

To obtain the potentially best and most robust coefficients, it is recommended that measurement pairs (temperature vs. capacitance) 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 ZSSC3230 output precision after calibration in order to avoid precision losses caused by the nominal reference values (that is capacitance signal and temperature deviations).

Note: There is an inherent redundancy in the seven capacitive 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 capacitive range, and then program it to the NVM configuration register Sensor_config (ZSSC3230). For data collection, the following two commands have to be used:

- For external sensor values:
  A2HEX: Single raw data capacitive sensor measurement for which the configuration is loaded from the *Sensor_config* register
- For temperature values:
  A6HEX: Single raw data temperature measurement for which the configuration register is loaded from an internal temperature configuration register (preprogrammed by IDT in NVM prior to IC delivery). This is the recommended approach for temperature data collection.

2.2.1.3 Raw Data Output

Depending on configured capacitive sensor resolution, the raw data measurement results are LSB (Least Significant Bit)-aligned. The internal temperature sensor has a preconfigured setup with an ADC resolution of 14-bit. Note that the capacitive raw output range is different from the temperature output range. Figure 4 summarizes the recommended raw data process before passing it to the *CalculateCoefficients* function of the DLL.

In order to adapt both capacitive and temperature raw values to the expected format (decimal representation, 24-bit, MSB-aligned in the range of -2^23..2^23 in).

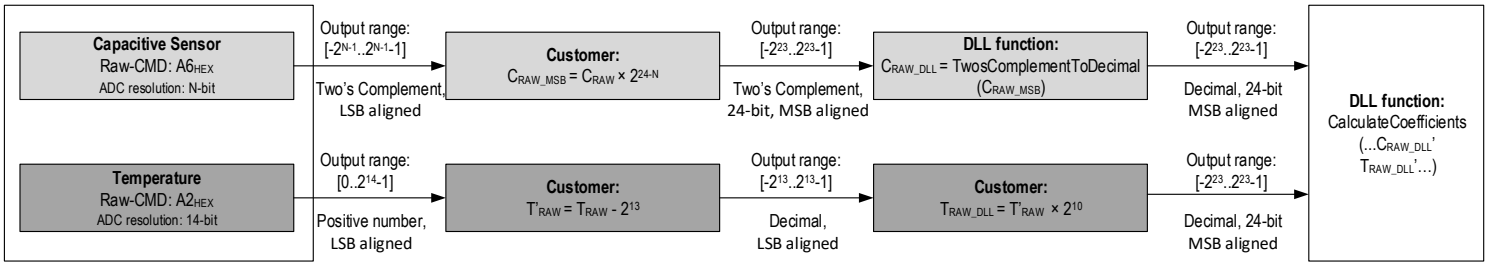
The capacitive sensor raw measurement data has to be:

- Multiplied by 2^(24-N), where N is the configured ADC-resolution. This has to be done by the customer.
- Converted from two's complement to decimal representation. For it the DLL function *TwosComplementToDecimal* can be used.

The temperature sensor raw measurement data has to be:

1. Subtracted by 2^13 to adjust the output range from 0..2^14-1 to -2^13..2^13-1.
2. Multiplied by 2^10 to bring the values from 14-bit to the 24-bit range.

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 CalibrationL6.DLLfor details). Features of the DLL are:

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

2.4 Programming NVM

The coefficients must be written to the NVM after they are calculated using the DLL. Table 2 lists the commands necessary to program the remaining coefficients (assumption: data collection via Raw Measurement Requests).

Table 2. Commands for Programming Coefficients and Final Settings of ZSSC3230

Command (Hex)	Data from Coefficients for the According Register	Description	Provided by
23	coefficients[INDEX_OFFSET_S] & 0x00FFFF	Offset_S[15:0]	DLL
24	coefficients[INDEX_GAIN_S] & 0x00FFFF	Gain_S[15:0]	DLL
25	coefficients[INDEX_TCG] & 0x00FFFF	Tcg[15:0]	DLL
26	coefficients[INDEX_TCO] & 0x00FFFF	Tco[15:0]	DLL
27	coefficients[INDEX_SOT_TCO] & 0x00FFFF	SOT_tco[15:0]	DLL
28	coefficients[INDEX_SOT_TCG] & 0x00FFFF	SOT_tcg[15:0]	DLL
29	coefficients[INDEX_SOT_S] & 0x00FFFF	SOT_sensor[15:0]	DLL
2A	coefficients[INDEX_OFFSET_T] & 0x00FFFF	Offset_T[15:0]	DLL
2B	coefficients[INDEX_GAIN_T] & 0x00FFFF	Gain_T[15:0]	DLL
2C	coefficients[INDEX_SOT_T] & 0x00FFFF	SOT_T[15:0]	DLL
2D	coefficients[INDEX_OFFSET_S] & 0x7F0000	Offset_S[22:16]	DLL
	coefficients[INDEX_GAIN_S] & 0x7F0000	Gain_S[22:16]	DLL
	coefficients[INDEX_OFFSET_S] & 0x800000	Offset_S[23]	DLL
	coefficients[INDEX_GAIN_S] & 0x800000	Gain_S[23]	DLL

<table>
<tr><th>Command (Hex)</th><th>Data from Coefficients for the According Register</th><th>Description</th><th>Provided by</th></tr>
<tr><td></td><td colspan="3">Data stream composition for MSB/SIGN register bits by the example of the offset and gain coefficients of the external sensor:

<pre>
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 = 0x2D;
//register data combination
data_0Dhex = 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_nvm(cmd, data);
</pre>
Numerical example:
<pre>
// 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)
</pre>
⇨ offset_s_msb = 0x07
⇨ sign_offset_s = 1
⇨ gain_s_msb = 0x59
⇨ sign_gain_s = 0
⇨ data_0Dhex = 34649 = 0x8759

[a]</td></tr>
<tr><td rowspan="5">2E</td><td>(coefficients[INDEX_TCG] & 0x7F0000) >> 8</td><td>Tcg[22:16]</td><td>DLL</td></tr>
<tr><td>(coefficients[INDEX_TCO] & 0x7F0000) >> 16</td><td>Tco[22:16]</td><td>DLL</td></tr>
<tr><td>(coefficients[INDEX_TCG]<0) ? 1 : 0</td><td>Tcg[23]</td><td>DLL</td></tr>
<tr><td>(coefficients[INDEX_TCO] <0) ? 1 : 0</td><td>Tco[23]</td><td>DLL</td></tr>
<tr><td colspan="3">data_0Ehex = register data combination as described in the example above in the example</td></tr>
</table>

Command (Hex)	Data from Coefficients for the According Register	Description	Provided by
2F	(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_0Fhex = register data combination as described in the example above in the example		
30	(coefficients[INDEX_SOT_S] & 0x7F0000) >> 8	SOT_sensor[22:16]	DLL
	(coefficients[INDEX_OFFSET_T] & 0x7F0000) >> 16	Offset_T[22:16]	DLL
	(coefficients[INDEX_SOT_S] <0) ? 1 : 0	SOT_sensor[23]	DLL
	(coefficients[INDEX_OFFSET_T] <0) ? 1 : 0	Offset_T[23]	DLL
	data_10hex = register data combination as described in the example above in the example		
31	(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_11hex = register data combination as described in the example above in the example		
32	Arbitrary, fix, from user 0xXXXX	Sensor_config (sensor specific AFE configuration for the external sensor measurement)	User

[a] The composition is equivalent for all further SIGN/MSB-registers.

2.5 Verification

The DLL interface provides verification at calibration time (see section 3.3.4). It is beneficial to perform an online verification at a different sensor measurand/temperature combination than which 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*. Therein should be the declarations for the functions used from *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

Take the following steps for calibration when using the CalibrationL6.DLL:

1. Data Conversion: All input raw and target data for both sensor and temperature (if applicable) must be converted into the correct format, see section 2.2.1.
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 3.3.4.

3.2.1 Using Customer Default Values as Coefficients

The *CalibrationL6.DLL* library supports calibration using customer-calculated default values. A customer default value can be applied to all calibrations without recalculation each time, which is 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 will always be less accurate than a complete calibration. To use a default value during calibration, do not select coefficient for calculation.

Note: Apply the customer default values at calculation time, not at data collection. For example, a customer default value of 1FHEX for Tcg is provided to the DLL, while 00HEX is programmed into the Tcg NVM field prior to data collection.

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. These enumerations and their use are described in this section.

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: Declare 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-combined 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 temperature coefficients
errorcode = CalculateCoefficients(  coefficients,
                                    &negCoeffs
                                    2,
                                    P3_S,
                                    0,
                                    raw_cap,
                                    desired_cap,
                                    raw_temp,
                                    desired_temp, /* Not calibrating anything with temp */
                                    );
```

3.3.1.3 Indices for Coefficients

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

C code declaration:

```
//INDICES 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 Capacitive Sensor (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 as sensor input are useful for defining the common range of the measured capacitance. For calculation or verification routines listed in sections 3.3.3 and 3.3.4, the sensor inputs must be processed through *ConvertBridgeFromPercent* routine which maps the sensor values in percentage [0, 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, the 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 will be 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 Passed to Capacitive Sensor (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: It returns an error code denoting the status of the calculations. A '0' is returned if the method was passed successfully. A '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 desired_temp_1;
int errorcode = 0;

errorcode = ConvertTempFromDegrees(&desired_temp_1, 50.0, -40.0, 125.0);
```

ConvertTempToDegrees can be used to convert a 24-bit temperature as returned by *GetCorrectedTemp* into degrees Celsius. This method can also be used to convert a promoted normal mode temperature reading.

C code declaration:

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

Returns: It returns an error code denoting the status of the calculations. A '0' is returned if the method was passed successfully. A '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 temp_1;
double tempC;
int errorcode = 0;

temp_1 = GetCorrectedTemp(coefficients, 320000);
errorcode += ConvertTempToDegrees(&tempC, temp_1, -40, 85);
```

Table 6 Parameter Passed to 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 decimal number in the range [-2^23..2^23-1].

Raw capacitance measurement results are provided from the ZSSC3230 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 decimal values, they have to be converted accordingly. Further details are described in section 2.2.1.3.

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 test_twoscomp = 0;
__int32 sign_magn = 0;

test_twoscomp = 0xffffff6;
sign_magn = TwosComplementToDecimal(test_twoscomp);
// sign_magn = -10

test_twoscomp = 0x7000A3;
sign_magn = TwosComplementToDecimal(test_twoscomp);
// sign_magn = 7340195

test_twoscomp = 0x5;
sign_magn = TwosComplementToDecimal(test_twoscomp);
// sign_magn = 5

test_twoscomp = 0x800005;
sign_magn = TwosComplementToDecimal(test_twoscomp);
// sign_magn = -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 3.3.4.

C code declaration:

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

Returns: The function returns an error code denoting the status of the calculations. A '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 rawCap[2], desiredCap[2];

// temperature input not relevant
double dummy_raw[2] = = {NULL,NULL};
double dummy_desired[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
rawCap[0] = -10000.0;
rawCap[1] = 8236410.0;

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

// run coefficients calculation
errorcode = CalculateCoefficients(  coefficients,
                                    &negCoeffs,
                                    numPoints,
                                    selCoeffs,
                                    calType,
                                    rawCap,
                                    desiredCap,
                                    dummy_raw,     /* Not calibrating anything with temp */
```

```
                                        dummy_desired  /* Not calibrating anything with temp */
                                                       );
/************resulting coefficients******
coefficients[0] = coefficients[INDEX_OFFSET_S] = -1028301
coefficients[1] = coefficients[INDEX_GAIN_S] = 3413303
error_code = 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.
*sensorRaw [a]	Array of raw sensor values. Must be converted for DLL input and have the length of numPoints. If not calibrating for capacitance correction, the array elements can be NULL.
*sensorDesired [a]	Array of target sensor values. Must be converted for DLL input and have the length of numPoints If not calibrating for capacitance 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 rawT);
```

Returns: The calibrated temperature in double-precision floating-point format is provided. It can be converted to Celsius using the *ConvertTempToDegree* function, see section 3.3.2.2.

3.3.4.2 GetCorrectedBridge

GetCorrectedBridge calculates the calibrated sensor 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 rawCap, double rawT);
```

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

Example: To verify that the accuracy at the calibration points is within 1.5% and 3°C, assuming a 3 point sensor calibration has been accomplished with data stored in rawCap[3], rawT[3], sensor[3], and temp[3]. Such verification does not include the inaccuracies caused by the sensor and measurement, but the deviations caused by correction calculation.

```
double rawCap[3], rawT[3];
double refCapCodes[3], refTempCodes[3];
double refCapPerc, refTempDeg;

int errorcode = 0;

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

double outCapCodes, outCapPerc, outTempCodes, outTempDeg;

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

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

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

        outCapCodes = GetCorrectedBridge(coefficients, rawCap[i], rawT[i]);
        outCapPerc = ConvertBridgeToPercent(outCapCodes);
        refCapPerc = ConvertBridgeToPercent(refCapCodes[i]);

        // check external sensor accuracy comparing percentage values
        // between measured and reference values
        if( fabs(outCapPerc-refCapPerc) > 1.5 ){…} //ERROR

};
```

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

Revision Date	Description of Change
October 18, 2019	Initial release