Application Note

How to Read and Interpret Digital Temperature Sensor Output Data

Ren Schackmann

ABSTRACT

Digital temperature sensors are now an industry-standard due to accuracy and compatibility with various digital interfaces such as I2C, SPI, UART, 1-Wire, PWM, and the emerging I3C MIPI. These plug-and-play devices require no signal conditioning.

At the core, a digital temperature sensor consists of a bias or band-gap reference, integrated or remote temperature sensing transistors, and an integrated analog-to-digital converter (ADC). Note that ADCs in temperature sensors come in different resolutions. For example, a 12-bit ADC output commonly has an LSB of 0.0625°C. After the ADC processes the sensor data, the raw outputs are sent through the digital interface and must be converted into temperature values. These outputs typically use a 2's complement signed fixed-point representation, which involves placing an implied binary point between two bit locations. This format maintains broad compatibility with various microcontrollers and processors, even those without floating-point support.

This application note provides an overview of the algorithm implementation using fixed-point mathematics. We use the 'Q Format' (also known as 'Q Notation' or 'Q Point') concept for fixed-point representation, to describe and differentiate typical output encodings for temperature sensors. The Qm.n labeling convention represents different temperature sensor output formats and encodings. These concepts are explained at a core quantitative level and then demonstrated using snippets of C code, JavaScript, Python and Microsoft® Excel.

Table of Contents

List of Tables

Trademarks

Microsoft® is a registered trademark of Microsoft Corporation.
Mozilla® is a registered trademark of Mozilla Foundation.
All trademarks are the property of their respective owners.

1 Introduction

2's Complement and Q format are both employed in the data encoding of TI temperature sensors. 2's Complement is a method of encoding negative numbers. Half of the available range (1-bit) is sacrificed to express whether the number is negative or not. Q format is a method of encoding rational numbers. Typically four or more bits are reserved to express fractional values between 1 and 0.

1.1 2's Complement

2's complement is a common method for encoding signed integers in computers, and this method is also employed in TI temperature sensors. Many programming languages can store and manipulate 2's complement data natively. The theory for encoding or decoding negative numbers in 2's complement is as follows:

1. Take the binary value and invert all the bits (for example, 1 to 0, 0 to 1)
2. Add 1 to the inverted number, remembering to carry overflow bits to the left

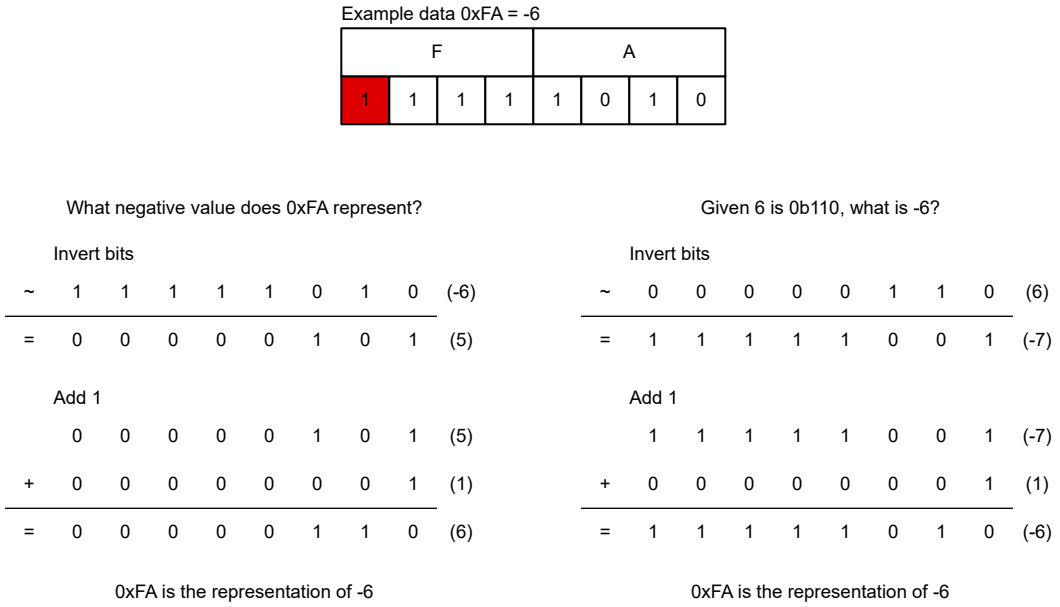

Figure 1-1. Applying 2's Complement procedure

1.1.1 2's Complement Traits

Table 1-1. 3-bit 2's Complement Table

Binary	Signed Value	Unsigned Value
0b100	-4	4
0b101	-3	5
0b110	-2	6
0b111	-1	7
0b000	0	0
0b001	1	1
0b010	2	2
0b011	3	3

This table provides the full range of values in a 3-bit 2's complement number. The low bit count enables us to easily see all the possible values and observe common traits in 2's Complement encoding.

Here are some facts about 2's complement to keep in mind:

- The most significant bit indicates sign.
- The **highest number** that can be expressed is 0 followed by all 1's in binary, which is 0b011 shown here.

- The **largest absolute value** that can be expressed is 1 followed by all 0's (0b100 here,) but this is **always a negative number**.
- **All 1's in binary is always equal to -1.**
- Adding +1 to -1 causes rollover to 0, which is the expected mathematical result.

In the following abbreviated table, we can see the same traits apply to an 8-bit 2's Complement encoding. Note that the column Signed Value describes the 8-bit C data type int8_t and the column Unsigned Value describes the 8-bit C data type uint8_t.

Table 1-2. 8-bit 2's Complement Table

Binary	Hex	Signed Value	Unsigned Value
0b10000000	0x80	-128	128
0b10000001	0x81	-127	129
...	...	...	...
0b11111101	0xFD	-3	253
0b11111110	0xFE	-2	254
0b11111111	0xFF	-1	255
0b00000000	0x00	0	0
0b00000001	0x01	1	1
0b00000010	0x02	2	2
...	...	...	...
0b01111110	0x7E	126	126
0b01111111	0x7F	127	127

1.2 Q Format

Q format is a method of encoding rational numbers. Typically four or more bits are reserved to express fractional values between 1 and 0. Rational data stored in Q format can be manipulated and stored efficiently without the need for floating point operations that are sometimes prohibited in microcontroller code. In this document, the number following Q refers to the number of fractional bits. Other sources denote the number of integer bits in addition to the fractional bits, as shown in the table, where *m* is the number of integer bits, including the sign, and *n* is the number of fractional bits. All sources reviewed agree on a shorthand *Qn* notation that only includes the *n* fractional bits after the Q, which is consistent with this document.

Table 1-3. Q Notation Variants

Source	Q Format	Q Example	Example Detail
This document	Qn	Q4	12 bits total 8 integer bits including sign (7 integer bits without sign) 4 fractional bits
Variant 1	Qm.n	Q8.4	
Variant 2 (TI)	Q(m-1).n	Q7.4	
Variant 3 (ARM)	Qn.m	Q4.8	

Q format can also be called a Fixed-point data format. Fixed-point data has a preconfigured resolution where floating point data has a variable resolution. The following table shows the relationship between resolution and bit weight for a chosen Q format. Higher Q formats are possible, but are not employed in temperature sensors today.

Table 1-4. Q format Scaling Factors

Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7
1	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125
1	1/2	1/4	1/8	1/16	1/32	1/64	1/128
2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}

As an example, integer data that is known to be in Q4 format can be converted to the rational Q value by multiplying the data with 0.0625, 1/16 or 2^{-4} because these values are all equal.

1.3 Common Temperature Data Format

Most digital temperature sensors, especially those with an I2C interface, utilize the 12 bit Q4 Format. The original LM75 sensor offers 9 bit resolution with Q1 Format. The LM75 sensor can only report in half degree increments with the single Q bit. Successors to the LM75 offered configurable 9/10/11/12 bit resolution. When enabled, these extra bits are Q bits, and offered Q1/Q2/Q3/Q4 formats respectively. Despite the extra bits, these formats are still 100% software compatible. The compatibility comes from the constant location of the Decimal point, shown in Figure 1-2; the output data is not shifted within the register when bits are missing.

One convenient feature of this format is that computing the temperature output can be greatly simplified if rational resolution is not needed. The upper eight bits of the result represent the whole number temperature, and don't need further calculation steps. See the example in the Figure below where the upper bits have a value of 32, and the temperature is 32.5625°C.

The caveat of this format is that the format can not express a temperature beyond 128°C, while modern sensors are rated for operation up to 150°C.

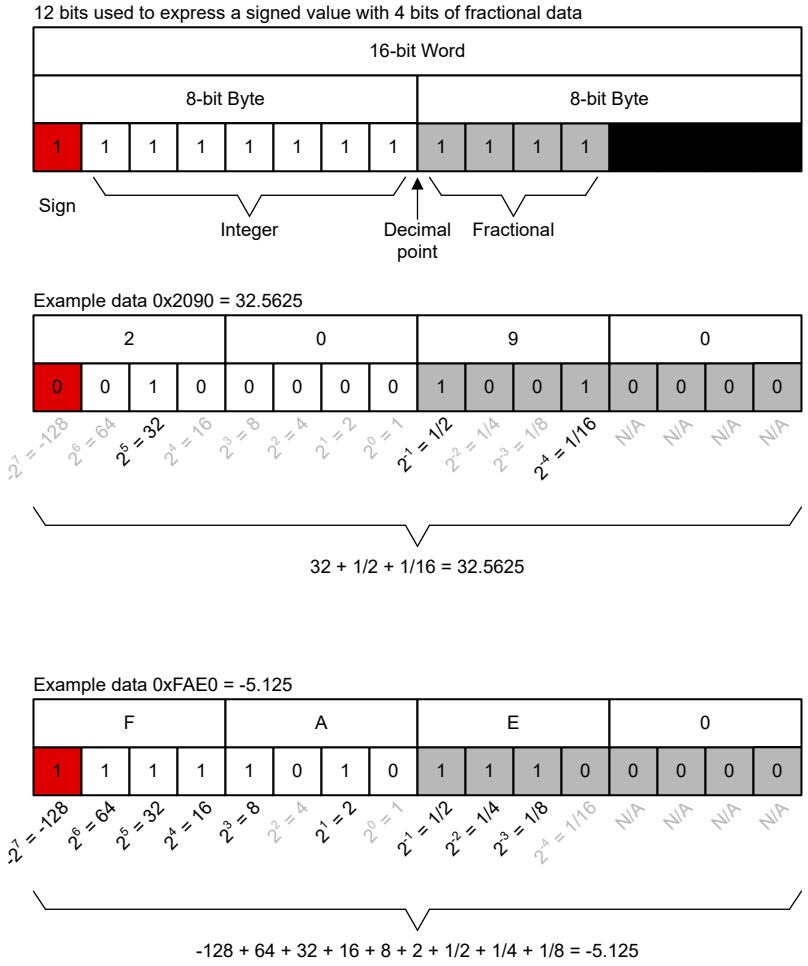

Figure 1-2. 12-bit Q4 Format

Table 1-5. 12-Bit Q4 Example Data

Temperature	Digital Output	
	Binary	Hex
127.9375°C	0111 1111 1111 0000	7FF0
125°C	0111 1101 0000 0000	7D00
25°C	0001 1001 0000 0000	1900
0.0625°C	0000 0000 0001 0000	0010
0°C	0000 0000 0000 0000	0000
−0.00625°C	1111 1111 1111 0000	FFF0
−25°C	1110 0111 0000 0000	E700
−40°C	1101 1000 0000 0000	D800

1.4 High Accuracy Temperature Data Format

Modern sensors, such as the TMP117, offer a full 16 bits of resolution in a Q7 format. Shifting the decimal point to the right by one bit (relative to the common format) allows this format to express a range of +255 to -256, although devices are not specified for the extremes. There are no unused bits in the lower byte, resulting in 0.0078125C resolution from the Q7 format.

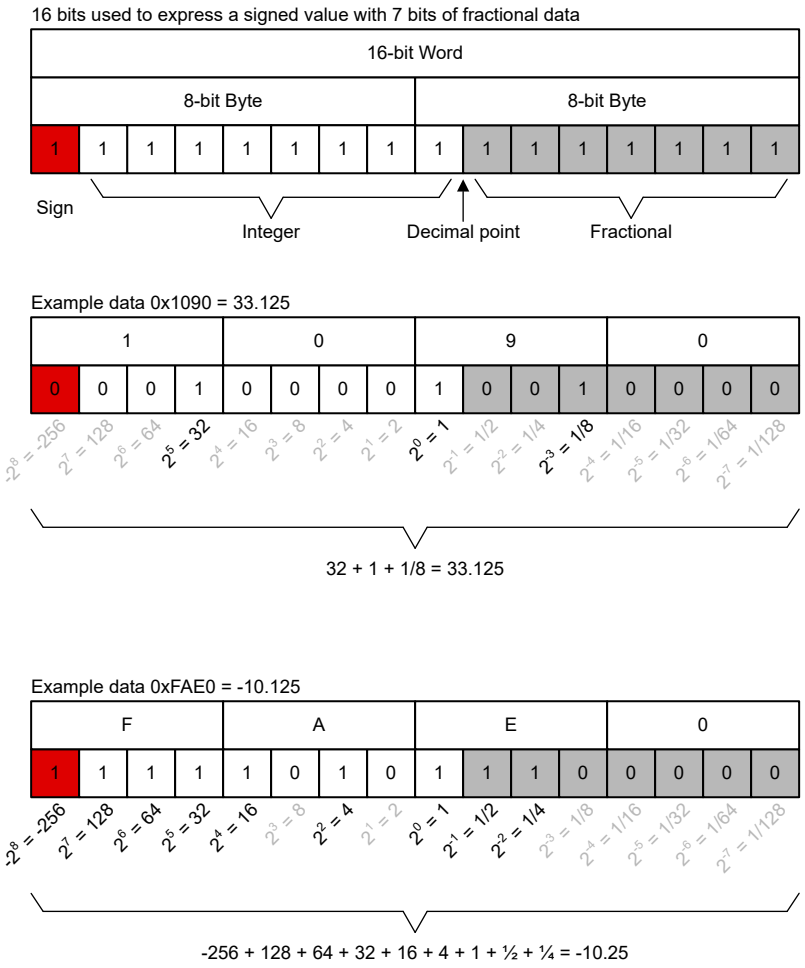

Figure 1-3. 16 bit Q7 Format

Table 1-6. 16-Bit Q7 Example Data

Temperature	Digital Output	
	Binary	Hex
+125°C	0011 1110 1000 0000	3E80
+25°C	0000 1100 1000 0000	0C80
+0.0078125°C	0000 0000 0000 0001	0001
0°C	0000 0000 0000 0000	0000
−0.0078125°C	1111 1111 1111 1111	FFFF
−25°C	1111 0011 1000 0000	F380
−40°C	1110 1100 0000 0000	EC00

2 Code Examples

This section provides ready-to-use C Code examples for existing TI products. For further information on programming, see also Section 3.

This section is divided down into groups of digital temperature sensors that have the same number of bits and Q format. Each section includes a table that includes the number of bits, Q format, resolution, temperature range, first byte temperature, and digital output at 25˚C. The **First Byte Integer C** row is a yes-or-no descriptor that means temperature is expressed as a whole number in the first byte (first 8 bits) so the second byte can be discarded if extra resolution is not desired.

2.1 16 Bits With Q7 Notation

Sensors: TMP114, TMP116, TMP117

2.1.1 Properties

Table 2-1. 16-Bit Q7 Encoding Parameters

Parameter	Value
Bits	16
Q	7
Resolution	0.0078125
Range (+)	255.9921875
Range (-)	-256
First Byte Integer C	No
25˚C	0xC80

Table 2-2. 16-Bit Q7 Bit Values

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sign	128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125
-256	128	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128
-2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}

2.1.2 C Code

```
/* 16-bit format will have 0 bits discarded by right shift
   q7 is 0.007812 resolution
   the following bytes represent 24.5C */
uint8_t byte1 = 0xC;
uint8_t byte2 = 0x40;
float f = ((int8_t) byte1 << 8 | byte2) * 0.0078125f;
int mC = ((int8_t) byte1 << 8 | byte2) * 1000 >> 7;
int C = ((int8_t) byte1 << 8 | byte2) >> 7;
```

2.2 12-bits With Q4 Notation

Sensors: TMP102, TMP112, TMP1075, TMP75, TMP75B, TMP75C, LM75, LM75B, TMP175, TMP275, TMP108, TMP144, TMP100, TMP101, TMP400, TMP421, TMP422, TMP423, TMP461

These devices have 12-bits with Q4 notation. Other devices that have these characteristics are the LM74 & any sensors ending in 75.

Several of these products can be configured for 9, 10, 11 or 12 bit output. The encoding and decoding is the same for each setting, and 12-Bit Q4 decoding can be applied to the output regardless of resolution configuration. This is because the bits that are not used do not contribute anything to the result.

2.2.1 Properties

Table 2-3. 12-Bit Q4 Encoding Parameters

Parameter	Value
Bits	12
Q	4
Resolution	0.0625

Table 2-3. 12-Bit Q4 Encoding Parameters (continued)

Parameter	Value
Range (+)	127.9375
Range (-)	-128
First Byte Integer C	Yes
25˚C	0x1900

Table 2-4. 12-Bit Q4 Bit Values

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sign	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	-	-	-	-
-128	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	-	-	-	-
-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	-	-	-	-

2.2.2 C Code

```
/* 12-bit format will have 4 bits discarded by right shift
   q4 is 0.062500 resolution
   the following bytes represent 24.5C */
uint8_t byte1 = 0x18;
uint8_t byte2 = 0x80;
float f = (((int8_t) byte1 << 8 | byte2) >> 4) * 0.0625f;
int mC = (((int8_t) byte1 << 8 | byte2) >> 4) * 1000 >> 4;
int C = (int8_t) byte1;
```

2.3 13-bits With Q4 Notation (EM=1)

Sensors: TMP102, TMP112, TMP144

This is a special case for the TMP102, TMP112. & TMP144 devices. The Extended Mode bit (EM = 1) makes these a 13-bit device rather than 12-bit. The notation is still Q4.

2.3.1 Properties

Table 2-5. 13-Bit Q4 Encoding Parameters

Parameter	Value
Bits	13
Q	4
Resolution	0.0625
Range (+)	255.9375
Range (-)	-256
First Byte Integer C	No
25˚C	0xC80

Table 2-6. 13-Bit Q4 Bit Values

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sign	128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	-	-	-
-256	128	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	-	-	-
-2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	-	-	-

2.3.2 C Code

```
/* 13-bit format will have 3 bits discarded by right shift
   q4 is 0.062500 resolution
   the following bytes represent 24.5C */
uint8_t byte1 = 0xC;
uint8_t byte2 = 0x40;
float f = (((int8_t) byte1 << 8 | byte2) >> 3) * 0.0625f;
```

```
int mC = (((int8_t) byte1 << 8 | byte2) >> 3) * 1000 >> 4;
int C = (((int8_t) byte1 << 8 | byte2) >> 3) >> 4;
```

2.4 13-bits With Q4 Notation

Sensors: TMP468, TMP464, TMP121, TMP122, TMP123, TMP124

2.4.1 Properties

Table 2-7. 13-Bit Q4 Encoding Parameters

Parameter	Value
Bits	13
Q	4
Resolution	0.0625
Range (+)	255.9375
Range (-)	-256
First Byte Integer C	No
25˚C	0xC80

Table 2-8. 13-Bit Q4 Bit Values

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sign	128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	-	-	-
-256	128	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	-	-	-
-2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	-	-	-

2.4.2 C Code

```
/* 13-bit format will have 3 bits discarded by right shift
   q4 is 0.062500 resolution
   the following bytes represent 24.5C */
uint8_t byte1 = 0xC;
uint8_t byte2 = 0x40;
float f = (((int8_t) byte1 << 8 | byte2) >> 3) * 0.0625f;
int mC = (((int8_t) byte1 << 8 | byte2) >> 3) * 1000 >> 4;
int C = (((int8_t) byte1 << 8 | byte2) >> 3) >> 4;
```

2.5 14-bits With Q6 Notation

Sensors: TMP107

2.5.1 Properties

Table 2-9. 14-Bit Q6 Encoding Parameters

Parameter	Value
Bits	14
Q	6
Resolution	0.015625
Range (+)	127.984375
Range (-)	-128
First Byte Integer C	Yes
25˚C	0x1900

Table 2-10. 14-Bit Q6 Bit Values

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sign	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125	0.015625	-	-
-128	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	1/32	1/64	-	-
-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	-	-

2.5.2 C Code

```
/* 14-bit format will have 2 bits discarded by right shift
   q6 is 0.015625 resolution
   the following bytes represent 24.5C */
uint8_t byte1 = 0x18;
uint8_t byte2 = 0x80;
float f = (((int8_t) byte1 << 8 | byte2) >> 2) * 0.015625f;
int mC = (((int8_t) byte1 << 8 | byte2) >> 2) * 1000 >> 6;
int C = (int8_t) byte1;
```

2.6 TMP182x Formats

Sensors: TMP1826, TMP1827

These devices have a 16-bit Q7 mode called Precision Format. These devices use a 12-bit Legacy Format by default. The 12-bit Legacy Format is actually 16-bit Q4 for the purposes of this document.

2.6.1 Properties

Table 2-11. TMP182x Encoding Parameters

Parameter	Value	Value
Format	Precision	Legacy
Bits	16	16 (12 effective)
Q	7	4
Resolution	0.0078125	0.0625
Range (+)	255.9921875	127.9375
Range (-)	-256	-128
First Byte Integer C	No	No
25˚C	0xC80	0x190

Table 2-12. TMP182x Precision Format Bit Values

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sign	128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125
-256	128	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128
-2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}

Table 2-13. TMP182x Legacy Format Bit Values

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sign	Sign	Sign	Sign	Sign	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625
-2048	1024	512	256	128	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16
-2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

2.6.2 C Code

```
/* 16-bit format will have 0 bits discarded by right shift
   q7 is 0.007812 resolution
   the following bytes represent 24.5C */
uint8_t byte1 = 0xC;
uint8_t byte2 = 0x40;
float f = ((int8_t) byte1 << 8 | byte2) * 0.0078125f;
int mC = ((int8_t) byte1 << 8 | byte2) * 1000 >> 7;
int C = ((int8_t) byte1 << 8 | byte2) >> 7;
```

2.7 14-bits with Q5 Notation

Sensors: TMP126, TMP127, LM73, LM95071

2.7.1 Properties

Table 2-14. 14-Bit Q5 Encoding Parameters

Parameter	Value
Bits	14
Q	5
Resolution	0.03125
Range (+)	255.96875
Range (-)	-256
First Byte Integer C	No
25˚C	0xC80

Table 2-15. 14-Bit Q5 Bit Values

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sign	128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125	-	-
-256	128	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	1/32	-	-
-2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	-	-

2.7.2 C Code

```
/* 14-bit format will have 2 bits discarded by right shift
   q5 is 0.031250 resolution
   the following bytes represent 24.5C */
uint8_t byte1 = 0xC;
uint8_t byte2 = 0x40;
float f = (((int8_t) byte1 << 8 | byte2) >> 2) * 0.03125f;
int mC = (((int8_t) byte1 << 8 | byte2) >> 2) * 1000 >> 5;
int C = (((int8_t) byte1 << 8 | byte2) >> 2) >> 5;
```

2.8 8-bits With No Q Notation

Sensors: TMP103, TMP104, TMP4718 (Local)

2.8.1 Properties

Table 2-16. 8-Bit Q0 Parameters

Parameter	Value
Bits	8
Q	0
Resolution	1
Range (+)	127
Range (-)	-128
First Byte Integer C	Yes
25˚C	0x19

Table 2-17. 8-Bit Q0 Bit Values

7	6	5	4	3	2	1	0
Sign	64	32	16	8	4	2	1
-128	64	32	16	8	4	2	1
-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

2.8.2 C Code

```
/* 8-bit format will not have byte2
   q0 is 1.000000 resolution
   the following byte represents 24C */
uint8_t byte1 = 0x18;
int C = (int8_t) byte1;
```

2.9 11-bits With Q3 Notation

Sensors: TMP4718 (Remote)

2.9.1 Properties

Table 2-18. 11-Bit Q3 Parameters

Parameter	Value
Bits	11
Q	3
Resolution	0.125
Range (+)	127.875
Range (-)	-128
First Byte Integer C	Yes
25˚C	0x1900

Table 2-19. 11-Bit Q3 Bit Values

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sign	64	32	16	8	4	2	1	0.5	0.25	0.125	-	-	-	-	-

Table 2-19. 11-Bit Q3 Bit Values (continued)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-128	64	32	16	8	4	2	1	1/2	1/4	1/8	-	-	-	-	-
-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	-	-	-	-	-

2.9.2 C Code

```
/* 11-bit format will have 5 bits discarded by right shift
   q3 is 0.125000 resolution
   the following bytes represent 24.5C */
uint8_t byte1 = 0x18;
uint8_t byte2 = 0x80;
float f = (((int8_t) byte1 << 8 | byte2) >> 5) * 0.125f;
int mC = (((int8_t) byte1 << 8 | byte2) >> 5) * 1000 >> 3;
int C = (int8_t) byte1;
```

2.10 Devices Without 2's Complement

These digital temperature sensors do not utilize 2's complement format to read in temperature values: TMP401, TMP411, TMP431, TMP432, TMP435, and TMP451. For this reason, the decode does not cast them into a signed type. The way that these devices express a negative temperature is by enabling a RANGE bit which adds 64C to the result. When RANGE is enabled, the decode must subtract 64, causing a raw value of 0 to become -64C output.

2.10.1 Properties

Table 2-20. 12-Bit Q4 Parameters

Parameter	Value
Bits	12
Q	4
Resolution	0.0625
Range (+)	127.9375
Range (-)	0
First Byte Integer C	Yes
25˚C	0x1900

Table 2-21. 12-Bit Q4 Bit Values

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	-	-	-	-
-	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	-	-	-	-
-	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	-	-	-	-

2.10.2 C Code

```
/* 12-bit format will have 4 bits discarded by right shift
   q4 is 0.062500 resolution
   the following bytes represent 24.5C
   there is no cast into signed type */
uint8_t byte1 = 0x18;
uint8_t byte2 = 0x80;
float f = ((byte1 << 8 | byte2) >> 4) * 0.0625f;
int mC = ((byte1 << 8 | byte2) >> 4) * 1000 >> 4;
int C =  byte1;
```

3 Other Programming Languages

This section describes in detail the code required to perform temperature conversion in C, Microsoft Excel, JavaScript and Python. The procedure is to parse incoming data, apply 2's complement, discard unused bits, and apply Q format. The full code examples for Excel and JavaScript are provided in Section 3.4. This code is written for 12-bit, Q4 devices and needs to be adapted to other devices, though the principles are the same.

3.1 Parsing

- When programming hardware, the data is often returned as two 8-bit bytes.
  - If the language has explicit type conversions, initially store these bytes as unsigned to better control the conversion to signed bytes. The behavior that must be avoided is called **sign-extend** of the lower byte.

```
/* C Signed Types */
unsigned char x = 0xFF;
signed char y = 0xFF;
/* x is treated as 255 and y as -1 */

/* C99 fixed width integer types */
uint8_t x = 0xFF;
int8_t y = 0xFF;
/* x is treated as 255 and y as -1 */
```

- In C, **sscanf()** parses strings (char*) including numbers that have 0x prefix. It's counterpart, **printf()**, uses the same format strings. The length modifier is a less commonly used feature of the format string, but this modifier can help to clean up our small data types. Here is a table of data types and format strings with relevant length modifiers. Note that %i used in scanf detects and correctly decodes the 0x prefix while %d and others do not.

Table 3-1. C Format Strings With Length Modifier

Bits	Data Type	Fixed Width Type	Format String
8	char	int8_t / uint8_t	%hhi
16	short int	int16_t / uint16_t	%hi
32	int	int32_t / uint32_t	%i

```
/* C Parsing and Outputting Hex */
char *s = "0xFF";
uint8_t x;
sscanf(s, "%hhi", &x);
/* x is 255 */

/* printf without length modifier */
printf("%i, %d, %u, %x\n", x, x, x, x);
/* "-1, -1, 4294967295, ffffffff" is printed due to coercion into 32 bit types and sign-extend */

/* printf with length modifier */

printf("%hhu, 0x%hhX, %#hhx\n", x, x, x);
/* the desired "255, 0xFF, 0XFF" is printed */
```

- Hex is convenient to use. Excel has **HEX2DEC** and related functions. This function does not allow prepending of "0x." HEX2DEC turns "FFFFFFFFFF" (ten Fs) into -1, so this function is internally behaving as a 40-bit signed data type. This is plenty of bits for temperature data.
- Excel's binary functions, such as **DEC2BIN**, are limited to 9 bit functionality.
- While **HEX2DEC** is desired for decoding of temperature data, the complementary function **DEC2HEX** can help when encoding temperature data.
- In the following table, the result of the B column calculations are 15 and 0xA.

Table 3-2. Excel Example for Parsing

	A	B
1	F	=HEX2DEC(A1)
2	10	=DEC2HEX(A2)

- JavaScript, Python and C all accept numeral constants pre-fixed with 0x.
- JavaScript has the function **parseInt()** that parses "0x" notation for hex. For example, the parseInt() function correctly converts "0xA" into 10.
- JavaScript has the method **toString()** for string data, which creates a hex value if provided the argument 16.

```
/* JavaScript Parsing and Outputting Hex */
let x = 0xA
let y = parseInt("0xA")
let z = (10).toString(16)
let s = "0x" + x.toString(16).toUpperCase().padStart(2,'0')
/* x and y are 10, z is 'a' and s is '0x0A' */
```

- Python's **int()** converts strings if the base argument is provided.
- Python's **hex()** converts numbers to hexadecimal strings.

```
# Python Parsing and Outputting Hex
x = 0xA
y = int("A",base=16)
z = hex(10)
# x and y are 10, z is '0xa'
```

3.2 2's Complement

- In a language with explicit type conversions, 2's Complement is handled automatically when casting into the correct data type.

```
/* C Type Casting */
unsigned char x = 0xFF;
signed char y = x;
signed int z = (signed char) x;
/* x is 255 but both y and z are -1 */

/* C99 fixed width integer types */
uint8_t x = 0xFF;
int8_t y = x;
int8_t z = (int8_t) x;
/* x is 255 but both y and z are -1 */
```

- Without type conversion, an **IF** statement is needed to detect and correct numbers that are supposed to be negative.
  - For a 2's complement number with n bits, values greater than or equal to $2^{(n-1)}$ are actually negative numbers. Restated, if the $2^{(n-1)}$ bit is present, then the number is negative. This bit is also known as the sign bit. For an 8-bit number, the sign bit is equal to 0x80 or 0b10000000.
  - To decode negative values, subtract 2^n. This is needed to reverse the order of 2's complement negative values. See also Table 1-1.
    - The opposite is also true: when encoding a negative number to a signed binary data type, 2^n can be added to obtain the correct hex/binary value.
    - Note that adding or subtracting 2^n this way assumes that the underlying data type does not overflow. This is a safe assumption most of the time, because that is not necessary to correct the sign of the data this way if we can type cast to the correct number of bits instead. The underlying data type is likely a C float or similar with at least 32 bits.
  - In Excel, use the **IF** function and 2^n statements for readability. Excel does not have good support for bitwise operations. Subtract 2^n to convert a number to the correct negative value. These examples are for 16-bit numbers.
- In the following Excel example, row 1 illustrates decoding of data received from the sensor while row 2 illustrates encoding data to be sent to the device.

Table 3-3. Excel Example for 2's Complement

	A	B	C
1	FFFF	=HEX2DEC(A1)	=IF(B1>=2^15,B1-2^16,B1)
2	-1	=IF(A2<0,A2+2^16,A2)	=DEC2HEX(B2)

Table 3-4. Excel Calculation Result for 2's Complement

	A	B	C
1	FFFF	65535	-1
2	-1	65535	FFFF

- In JavaScript and Python, both hex and bitwise operators are available.
  - 0x8000 and 0x10000 are equivalent to 2^{15} and 2^{16} without the use of additional operators.
  - Bitwise AND comparison can be used to check for the presence of a sign bit instead of greater-than-equal-to comparison.

```
/* JavaScript */
/* decode from 8-bit 2's complement */
function decode(x) {return (x & 0x8000) ? x-0x10000 : x}
let n = decode(0xFFFF)
/* n is -1 */

/* encode to 8-bit 2's complement */
function encode(x) {return (x < 0) ? x+0x10000 : x}
let n = encode(-1).toString(16)
/* n is 'ffff' */
```

```
# Python
# decode from 8-bit 2's complement
def decode(x): return x-0x10000 if (x & 0x8000) else x
n = decode(0xFFFF)
# n is -1

# encode to 8-bit 2's complement
def encode(x): return x+0x10000 if (x & 0x8000) else x
n = hex(encode(-1))
# n is '0xffff'
```

3.3 Discard Unused Bits

- Some devices have unused bits on the right (least-significant) side that must be discarded.
- Discarding is accomplished with right shift (>>) in languages that support bit shift operations.
- In situations where left shift is needed, or if sign-extend can't be avoided, you can mask unwanted bits to remove them. Masking is best accomplished with bitwise AND. The mask is a constant which has the bits that need to be retained set to logic 1. For example, a mask of 0xF is equivalent to 0b1111, and retains only the four least significant bits.

```
/* right shift */
unsigned char x = 0x23;
unsigned char y = x >> 4;
/* y is 0x02 */

/* mask to discard bits */
unsigned char z = x & 0xF;
/* z is 0x03 */
```

- In Excel, **multiply** and **divide** operations can be equivalent to left and right shift. For example, multiply times 2^3 is equivalent to left shift 3. Right shift can be accomplished with $*2^{-b}$ or $/2^b$. When right shifting this way, the result must be rounded to a whole number to discard the low bits. The INT() function is used here to round down.
- BITAND() can be used for masking, but beware this takes decimal input and provides decimal output. This can make BITAND() confusing to use for masking. BITAND() works with at least 32 bits, similar to HEX2DEC().
- The following Excel example demonstrates the same shift and discard operations on 0x23 as illustrated in the other language examples. Note that in this example, Excel is showing the decimal result in Column C. Our result coincidentally appears the same in decimal as the result does if displayed in hex as seen is in the other examples. Row 1 is the shift while row 2 is the discard operation.

Table 3-5. Excel Example for Discarding Bits

	A	B	C
1	23	=HEX2DEC(A1)	=INT(B1*2^-4)
2	23	=HEX2DEC(A2)	=BITAND(B2,HEX2DEC("F"))

Table 3-6. Excel Calculation Result for Discarding Bits

	A	B	C
1	23	35	2
2	23	35	3

- JavaScript and Python support shift as well as bitwise AND.

```
/* JavaScript */
/* right shift */
let x = 0x23
let y = x >> 4
/* y is 0x2 */

/* mask to discard bits */
let z = x & 0xF
/* z is 0x3 */
```

```
# Python
# right shift
x = 0x23
y = x >> 4
# y is 0x2

# mask to discard bits
z = x & 0xF
# z is 0x3
```

3.4 Apply Q format

- The value must be converted to a floating-point data type, and then multiplied by the device resolution.
- If a floating data type needs to be avoided, the fractional temperature data can be discarded using right shift. Alternatively, an integer data type can store mC (milliCelsius) data by taking the product of 1000 before the right shift.

```
/* C Decode 12-bit Q4 */
unsigned char b1 = 0xff;
unsigned char b2 = 0xf0;

/* combine 8 bit bytes into 16 bit word,
 apply signed type cast to upper byte */
int x = (signed char) b1 << 8 | b2;

/* shift to discard unused bits */
int y = x >> 4;

/* Q4 is 2^-4 = 1/16 = 0.0625
 cannot use shift operators on float
 use multiply or divide to create right shift */
float a = y * 0.0625f;

/* discard Q4 bits for a whole number result */
int b = y >> 4;

/* scale by 1000 then shift by Q# to get
 fractional result without float data type */
int c = y * 1000 >> 4; /* milliCelsius */

/* a is -0.0625, b is -1, and c is -63 */
printf("x:%d y:%d a:%f b:%d c:%d\n",x,y,a,b,c);
```

```
/* C Encode 12-bit Q4 */
float in = -0.0625f;
```

```
/* Q4 is 2^-4 = 1/16 = 0.0625
 cannot use shift operators on float
 emulate left shift using multiply */
short int r = in * 16;

/* left shift to create unused/discard bits */
short int s = r << 4;

/* s is 0xFFF0 */
printf("r:%d s:%d sx:%hx",r,s,s);
```

- Table 3-7 shows a complete Excel design for 12-bit, Q4. Row 1 illustrates decoding of data received from the sensor while row 2 illustrates encoding data to be sent to the device.

Table 3-7. Excel Example for Q format

	A	B	C	D	E
1	1810	=HEX2DEC(A1)	=IF(B1>=2^15,B1-2^16,B1)	=INT(C1*2^-4)	=D1*0.0625
2	-0.0625	=IF(A2<0,A2+2^8,A2)	=INT(B2*2^4)	=C2*2^4	=DEC2HEX(D2)

Table 3-8. Excel Calculation Result for Q format

	A	B	C	D	E
1	1810	6160	6160	385	24.0625
2	-0.0625	255.9375	4095	65520	FFF0

The following code shows a complete JavaScript and Python decode design for 12-bit, Q4 that can be used to read temperature data.

```
/* JavaScript */
function decode(x) {return (((x & 0x8000) ? x - 0x10000 : x) >> 4) * 0.0625}
let x = decode(0x1810)
let y = decode(0xFFF0)
/* x is 24.0625 and y is -0.0625 */
```

```
# Python
def decode(x): return ((x-0x10000 if (x & 0x8000) else x) >> 4) * 0.0625
x = decode(0x1810)
y = decode(0xFFF0)
# x is 24.0625
# y is -0.0625
```

The following code shows temperature encoding JavaScript and Python design for 12-bit Q4 that can be used to encode temperature limit or offset settings.

```
/* JavaScript */
function encode(x) {return ((x < 0 ? x + 0x100 : x) * 16) << 4}
let x = encode(24.0625).toString(16)
let y = encode(-0.0625).toString(16)
/* x is '1810' and y is 'fff0' */
```

```
# Python
def encode(x): return int((x+0x100 if (x < 0) else x) * 16) << 4
x = hex(encode(24.0625))
y = hex(encode(-0.0625))
# x is '0x1810'
# y is '0xfff0'
```

4 Summary

Texas Instruments offers a wide variety of digital temperature sensors that are able to take values from an ADC and convert these values into degrees Celsius. This process is performed by following the 2's complement methodology and considering the bit size and Q format for each device. The code included for each section in this application note shows an example of how the device is able to perform each operation along with a table highlighting some key characteristics that explain why the device works the way the device does. For more information on device applications and layout recommendations, please refer to each individual device data sheet.

5 References

For more information on the devices referenced in this application note, see also:

- Texas Instruments, *Digital temperature sensors*, product page

For resources on computing and notation mentioned in this application note, see also the following resources:

- Texas Instruments, *TMS320C64x DSP Library Programmer's Reference*, user's guide
- Cornell University, *Two's Complement*, reference site
- Wikipedia, *Fixed-point arithmetic*, article
- Wikipedia, *Q (number format)*, article
- Wikibooks, *Floating Point Unit*, article
- ARM, *RealView Development Suite AXD and armsd Debuggers Guide*, user's guide
- Intel, *Reference Manual for Intel® Integrated Performance Primitives for Microcontrollers*, user's guide

For resources on specific programming languages used to convert temperature data, see also the following resources:

- Microsoft®, *HEX2DEC function*, support article
- Mozilla® Corporation, *parseInt()*, JavaScript programming reference
- Wikipedia, *C data types*, article
- Wikipedia, *Operators in C and C++*, article

6 Appendix: Q App Source Code

This utility can be used for temperature conversion of any Q format encoding. This code is written and tested for a Unix-like shell. We recommend to save the source code as q.c and compile the code using cc q.c -o qapp.

```
#include <stdio.h>
#include <unistd.h>

void main(int argc, char *argv[])
{
    int c;
    uint8_t bits = 16;
    uint8_t qnum = 7;
    uint8_t byte1;
    uint8_t byte2;
    int16_t data;
    int mode = 0;
    while ((c = getopt(argc, argv, "b:q:hx")) != -1)
        switch (c)
        {
        case 'b':
            sscanf(optarg, "%hhi", &bits);
            break;
        case 'q':
            sscanf(optarg, "%hhi", &qnum);
            break;
        case 'h':
            mode = 1; /* print help */
            break;
        case 'x':
            mode = 2; /* print example code */
        }
    uint8_t shift = 16 - bits;
    float resolution = (float)1 / (1 << qnum);
    /* recommend using a constant for resolution in application code */
    if (mode == 0)
    { /* parse byte(s) and print celsius */
        switch (argc - optind)
        {
        case 2:
            sscanf(argv[optind], "%hhi", &byte1);
            sscanf(argv[optind + 1], "%hhi", &byte2);
            data = (int8_t)byte1 << 8 | byte2;
            printf("input: %d (0x%hhX) and %d (0x%hhX) becomes %d (0x%hX)\n", byte1, byte1, byte2,
byte2, data, data);
            break;
        case 1:
            sscanf(argv[optind], "%hi", &data);
            printf("input: %d (0x%hX) is assumed to be 16-bit\n", data, data);
            break;
        case 0:
            data = 0x7FFF;
            printf("demo: %d (0x%hX)\n", data, data);
            break;
        }
        printf("%d-bit format will have %d bits discarded by right shift\n", bits, shift);
        data >>= shift;
        printf("q%d is %f resolution\n", qnum, resolution);
        float f = data * resolution;
        printf("float: %f C\n", f);
        /* preserve fractional result while using int only */
        int mC = data * 1000 >> qnum;
        printf("int: %d mC\n", mC);
        /* discard fractional result while using int only */
        int C = data >> qnum;
        printf("int: %d C\n", C);
    }
    else if (mode == 1) /* print help */
    {
        printf("qapp [-b bits] [-q qnum] [byte] [byte2]\n\
use -b to specify number of bits\n\
use -q to specify Q format, which describes resolution\n\
byte can be 16 bit data or the first of two 8 bit bytes\n\
byte2 is the second 8 bit byte that will be assembled with byte\n\
\n\
Device settings:\n\
TMP117, TMP114:\n\
```

```
 qapp -b 16 -q 7 0x0C80\n\
TMP102, TMP112, TMP1075:\n\
 qapp -b 12 -q 4 0x1900\n\
TMP102, TMP112 EM=1:\n\
 qapp -b 13 -q 4 0x0C80\n\
TMP468:\n\
 qapp -b 13 -q 4 0x0C80\n\
TMP107:\n\
 qapp -b 14 -q 6 0x1900\n\
TMP126:\n\
 qapp -b 14 -q 5 0x0C80\n\
");
    }
    else if (mode == 2) /* print example code */
    {
        /* use 24.5 to represent nominal, ambient temperature with the first fractional bit set */
        int16_t exdata = 24.5f / resolution;
        exdata <<= shift;
        printf("C Code Examples:\n");
        printf("/* %d-bit format will have %d bits discarded by right shift\n", bits, shift);
        printf("   q%d is %f resolution\n", qnum, resolution);
        printf("   the following bytes represent 24.5C */ \n");
        printf("uint8_t byte1 = 0x%hhX;\n", exdata >> 8);
        printf("uint8_t byte2 = 0x%hhX;\n", exdata);
        if (shift)
            printf("float f = (((int8_t) byte1 << 8 | byte2) >> %d) * %gf;\n", shift, resolution);
        else
            printf("float f = ((int8_t) byte1 << 8 | byte2) * %gf;\n", resolution);
        if (shift)
            printf("int mC = (((int8_t) byte1 << 8 | byte2) >> %d) * 1000 >> %d;\n", shift, qnum);
        else
            printf("int mC = ((int8_t) byte1 << 8 | byte2) * 1000 >> %d;\n", qnum);
        if (shift + qnum == 8)
            printf("int C = (int8_t) byte1;\n");
        else if (shift)
            printf("int C = (((int8_t) byte1 << 8 | byte2) >> %d) >> %d;\n", shift, qnum);
        else
            printf("int C = ((int8_t) byte1 << 8 | byte2) >> %d;\n", qnum);
    }
}
```

7 Appendix: Device Summary Table

Device	Bits	Q	Resolution	Range+	Range-	First Byte	25C
TMP117	16	7	0.0078125	255.9921875	-256	No	0x0C80
TMP116	16	7	0.0078125	255.9921875	-256	No	0x0C80
TMP114	16	7	0.0078125	255.9921875	-256	No	0x0C80
TMP102	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP112	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP1075	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP75	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP75B	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP75C	12	4	0.0625	127.9375	-128	Yes	0x1900
LM75	12	4	0.0625	127.9375	-128	Yes	0x1900
LM75B	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP175	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP275	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP108	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP144	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP100	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP101	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP400	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP421	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP422	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP423	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP461	12	4	0.0625	127.9375	-128	Yes	0x1900
TMP102, EM=1	13	4	0.0625	255.9375	-256	No	0x0C80
TMP112, EM=1	13	4	0.0625	255.9375	-256	No	0x0C80
TMP144, EM=1	13	4	0.0625	255.9375	-256	No	0x0C80
TMP468	13	4	0.0625	255.9375	-256	No	0x0C80
TMP464	13	4	0.0625	255.9375	-256	No	0x0C80
TMP121	13	4	0.0625	255.9375	-256	No	0x0C80
TMP122	13	4	0.0625	255.9375	-256	No	0x0C80
TMP123	13	4	0.0625	255.9375	-256	No	0x0C80
TMP124	13	4	0.0625	255.9375	-256	No	0x0C80
TMP107	14	6	0.015625	127.984375	-128	Yes	0x1900
TMP1826 (Precision)	16	7	0.0078125	255.992187	-256	No	0x0C80
TMP1827 (Precision)	16	7	0.0078125	255.9921875	-256	No	0x0C80
TMP1826 (Legacy)	16 (12)	4	0.0625	127.9375	-128	Yes	0x190
TMP1827 (Legacy)	16 (12)	4	0.0625	127.9375	-128	Yes	0x190
TMP126	14	5	0.03125	255.96875	-256	No	0x0C80
TMP127	14	5	0.03125	255.96875	-256	No	0x0C80
LM73	14	5	0.03125	255.96875	-256	No	0x0C80
LM95071	14	5	0.03125	255.96875	-256	No	0x0C80
TMP103	8	0	1	127	-128	Yes	0x19
TMP104	8	0	1	127	-128	Yes	0x19

Device	Bits	Q	Resolution	Range+	Range-	First Byte	25C
TMP4718 (Local)	8	0	1	127	-128	Yes	0x19
TMP4718 (Remote)	11	3	0.125	127.875	-128	Yes	0x1900
TMP401	12	4	0.0625	127.9375	0	Yes	0x1900
TMP411	12	4	0.0625	127.9375	0	Yes	0x1900
TMP431	12	4	0.0625	127.9375	0	Yes	0x1900
TMP432	12	4	0.0625	127.9375	0	Yes	0x1900
TMP435	12	4	0.0625	127.9375	0	Yes	0x1900
TMP451	12	4	0.0625	127.9375	0	Yes	0x1900

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265