AND9213/D

Reading Smart Passive Sensors

ON Semiconductor®

www.onsemi.com

APPLICATION NOTE

Overview

ON Semiconductor's Smart Passive Sensors powered by a Magnus®−S integrated circuit (IC) offer up to three unique pieces of information to the reader: a Sensor Code, which is an indicator of the impedance seen at its RF input, an On−Chip RSSI Code, which indicates how much power the sensor tag is receiving from the reader, and a Temperature Code, which can be converted to an accurate reading of the temperature at the die. This note describes the procedures for reading these codes.

Determining the Sensor Tag Model Number

The Sensor, On−Chip RSSI, and Temperature Codes are stored at three different word addresses in the sensor tag memory, with the addresses depending on the Tag Model Number of the Magnus−S IC being used. If the Tag Model Number is not known in advance, it can be determined by reading word 1_h (hexadecimal) in the TID memory bank (bank 2_h) using a standard Class−1 Generation−2 UHF Read command.

The Tag Model Number will be four hexadecimal digits (all words in the sensor tag memory are two bytes wide). The three most significant digits determine the addresses of the sensor data. For example, if $402E_h$ is retrieved, the sensor data can be found in the tables below on the row with 402_h in the left column.

Reading the Sensor Code

The Sensor Code can be read from a tag using a standard Class−1 Generation−2 UHF Read command. The memory location depends on the Tag Model Number as described above and is given in Table 1.

Table 1. LOCATION OF SENSOR CODE VALUE

Tag Model Number Starts with	Memory Bank	Word Address
401_h	USER (Bank 3_h)	B_h
402_h	RESERVED (Bank 0_h)	B_h
403_h	RESERVED (Bank 0_h)	C_h

The Sensor Code is a 5- or 9-bit value, depending on the Tag Model Number. The remaining bits in the word are zero (Table 2).

Table 2. NUMBER OF BITS USED IN SENSOR CODE

Tag Model Number Starts with	Number of Bits Used in Sensor Code
401_h, 402_h	5
403_h	9

Reading the On−Chip RSSI Code

The On−Chip RSSI Code is a 5-bit value (decimal range from 0 to 31) indicating the amount of power the sensor tag is receiving. Larger numbers indicate higher received power levels. The On−Chip RSSI Code can be read with a two-step process:

1. Send a standard Class−1 Generation−2 UHF Select command to alert all sensor tags with an On−Chip RSSI code greater than, or less than/equal to a specified threshold
2. Send a standard Class−1 Generation−2 UHF Read command to retrieve the specific On−Chip RSSI Code for a particular sensor tag which satisfies the power threshold criterion.

Because of this process, it is possible to filter sensor tags according to the power they are receiving, silencing those tags with power above or below a particular threshold for the remainder of the inventory round. It is possible to have sensor tags respond regardless of their received power, which will be described later in an example.

May, 2015 − Rev. 1
Publication Order Number:
AND9213/D

The On−Chip RSSI Code is generated once the sensor tag receives a Select command with the parameters as described in Table 3. The value is stored in sensor tag volatile memory regardless of whether it is above or below the threshold, and remains until the reader signal turns off. But to respond to the subsequent Read command, the Mask parameter (Table 4) must agree with the On−Chip RSSI Code value.

Table 3. ON−CHIP RSSI SELECT COMMAND PARAMETERS

Tag Model Number Starts with	Memory Bank	Pointer Bit Address	Mask Length	Mask (Table 4)
401_h, 402_h	USER (Bank 3_h)	$A0_h$	8_h	M[7:0]
403_h	USER (Bank 3_h)	$D0_h$	8_h	M[7:0]

Table 4. BIT MASK FOR ON−CHIP RSSI SELECT COMMAND

Mask Bit	M7	M6	M5	M4	M3	M2	M1	M0
Bit Value	0	0	0: Match if Code is ≤ Threshold 1: Match if Code is > Threshold	5-bit Threshold Most Significant Bit First				

For example, to guarantee that the mask will match the sensor tag, regardless of its received power, mask bit 5 can be set to 0 and the threshold set to the maximum value of 11111, resulting in an 8-bit mask of 00011111 ($1F_h$). If the sensor tag satisfies the Select, the On−Chip RSSI can be retrieved with a Read command with the address given in Table 5. The Code is in the least-significant 5 bits of the word; the other bits in the word will be zero.

Table 5. LOCATION OF ON−CHIP RSSI CODE VALUE

Tag Model Number Starts with	Memory Bank	Word Address
401_h	USER (Bank 3_h)	9_h
402_h, 403_h	RESERVED (Bank 0_h)	D_h

Reading the Temperature Code

Magnus−S3 ICs are available with a precise temperature sensor. The sensor generates a Temperature Code which can be translated to a value in degrees. This feature is only available on Magnus−S ICs with a Tag Model Number starting with 403_h.

As with the On−Chip RSSI Code, reading the Temperature Code is a two-step process requiring standard UHF Select and Read commands.

1. Send a standard Class−1 Generation−2 UHF Select command with the parameters described in Table 6 to initialize the temperature sensor and calculate a Temperature Code.
2. Send a standard Class−1 Generation−2 UHF Read command to retrieve the Temperature Code from the sensor tag memory at the location given in Table 7.

Table 6. TEMPERATURE CODE SELECT COMMAND PARAMETERS

Tag Model Number Starts with	Memory Bank	Pointer Bit Address	Mask Length	Mask
403_h	USER (Bank 3_h)	$E0_h$	0_h	Empty

After the sensor tag has received the Select command, the Temperature Code will be available for reading in the memory location given in Table 7 below. The Temperature Code occupies the least-significant 12 bits of the word; the other bits are 0.

Table 7. LOCATION OF TEMPERATURE CODE VALUE

Tag Model Number Starts with	Memory Bank	Word Address
403_h	RESERVED (Bank 0_h)	E_h

Achieving an accurate Temperature Code requires the Select command to be followed by 2 ms of continuous wave before the reader issues any further commands. This pause gives the temperature sensor circuit time to run. The reader must not power down at any time between the Select and Read commands.

Calibrated Temperature Measurements

The Temperature Code can be converted to a precise temperature measurement by scaling it according to calibration data that are unique for each sensor tag. Temperature-enabled Magnus−S3 ICs come with calibration data pre-loaded in the last four words of the User memory bank (words 8_h, 9_h, A_h, and B_h). These four words – 64 bits – are organized into six data fields which describe a 2-point linear calibration. The fields are summarized in Table 8, and their locations in memory are mapped in Table 9, where it can be seen that the fields cross word boundaries.

Table 8. ORGANIZATION OF TEMPERATURE CALIBRATION DATA

Field Name	Starting Bit Number (MSB)	Number of Bits	Description
CRC	80_h	16	CRC−16 Applied to the Remaining 48 Calibration Data Bits
CODE1	90_h	12	Temperature Code Measured at the First Calibration Temperature
TEMP1	$9C_h$	11	(First Calibration Temperature ni Decimal Degrees C) × 10 + 800
CODE2	$A7_h$	12	Temperature Code at the Second Calibration Temperature
TEMP2	$B3_h$	11	(Second Calibration Temperature ni Decimal Degrees C) × 10 + 800
VER	BE_h	2	Calibration Format Version Number (Set to 0_h)

Table 9. LOCATION OF TEMPERATURE CALIBRATION DATA IN USER BANK

Word Address	Label	Bit Description															
8_h	Bit Address (Hex)	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
	Filed Description	CRC[15:0]															
9_h	Bit Address (Hex)	90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
	Filed Description	CODE1[11:0]												TEMP1[10:7]			
A_h	Bit Address (Hex)	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
	Filed Description	TEMP1[6:0]							CODE2[11:3]								
B_h	Bit Address (Hex)	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
	Filed Description	CODE2[2:0]			TEMP2[10:0]											VER[1:0]	

The CODE1 and TEMP1 fields describe the Temperature Code and actual temperature measured at the first calibration point. The CODE2 and TEMP2 fields describe the Temperature Code and temperature at the second point. The two points specify the linear response of the temperature sensor and are used to calculate a calibrated temperature at all other measured Temperature Codes.

The VER field stores a 2-bit version code and is set to 0_h. The CRC field contains a 16-bit CRC computed over the other 48 bits in the calibration data and follows the same specification as the CRC−16 defined in the EPC™ Generation−2 UHF RFID Specification, Annex F.2.

The TEMP1 and TEMP2 fields are 11-bit unsigned integers. They can be converted into temperatures in degrees Celsius by applying the following formula below to the TEMP1 or TEMP2 fields, expressed in decimal.

$$\text{Calibration Temperature in } ^\circ\text{C} = \frac{\text{TEMPx} - 800}{10} \quad \text{(eq. 1)}$$

To convert an arbitrary Temperature Code C into a calibrated value in degrees Celsius, apply the formula below, where all values are in decimal.

$$\text{Temperature in } ^\circ\text{C} = \frac{1}{10} \cdot \left[\left(\frac{\text{TEMP2} - \text{TEMP1}}{\text{CODE2} - \text{CODE1}} \cdot (\text{C} - \text{CODE1}) \right) + \text{TEMP1} - 800 \right] \quad \text{(eq. 2)}$$

Temperature Measurement Example

Suppose words 8_h, 9_h, A_h, and B_h in the User memory bank are read to be $BD9F_h$, $88A7_h$, $E147_h$, and 7900_h, respectively. Although it is optional, it is a good idea to verify that the CRC word agrees with the data. When $88A7E1477900_h$ is given to the CRC−16 algorithm, the CRC result that is generated is $BD9F_h$. This agrees with the CRC word stored in the calibration data, so the calibration information is valid. Example C# code for generating the CRC word is given in the Sample Code section of this document.

When the remaining three words are unpacked, it is seen that CODE1 = $88A_h$, TEMP1 = $3F0_h$, CODE2 = $A3B_h$, TEMP2 = 640_h, and VER=0_h (Figure 1). In decimal, these values are CODE1 = 2186, TEMP1 = 1008, CODE2 = 2619, TEMP2 = 1600, and VER = 0.

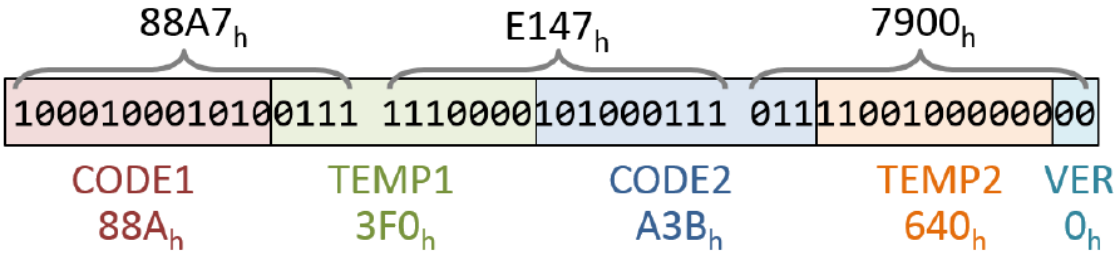

Figure 1. Unpacking Calibration Words into Fields

Suppose a temperature measurement is made and the Temperature Code reported is 2315 in decimal. This value, along with the other relevant calibration field values is plugged into the formula above to find the temperature.

$$\text{Temperature in } ^\circ\text{C} = \frac{1}{10} \cdot \left[\left(\frac{1600 - 1008}{2619 - 2186} \cdot (2315 - 2186) \right) + 1008 - 800 \right] = 38.44^\circ\text{C} \quad \text{(eq. 3)}$$

It is also possible to determine the temperature at which the first calibration point was taken by simply subtracting 800 from the decimal TEMP1 value and dividing by 10:

$$\frac{1008 - 800}{10} = 20.8^\circ\text{C} \quad \text{(eq. 4)}$$

Sample Code

The sample C# code below illustrates how to read the Sensor and On−Chip RSSI Code using ThingMagic RFID readers and the ThingMagic Mercury API.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ThingMagic;

namespace ThingMagicSample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Connect to the reader and adjust its settings ========================================================================
            Reader r = Reader.Create("tmr:///COM5");                    // Create Reader object
            r.Connect();                                                // Connect to the reader on emulated serial port COM5
            r.ParamSet("/reader/region/id", Reader.Region.NA);          // Set the regulatory region to North America
            r.ParamSet("/reader/radio/readPower", 2000);                // Set the reader power to 20 dBm

            // Read the Sensor Code ================================================================================================
            // Define a tag read operation which reads from a Magnus-S2;
            // Sensor Code is in the RESERVED bank, in memory location hex 8
            TagOp sensorCodeRead = new Gen2.ReadData(Gen2.Bank.RESERVED, 0x8, 1);
            // Define a read plan for antenna 1 using Gen2 protocol which uses our Sensor Code read operation
            SimpleReadPlan readPlan = new SimpleReadPlan(new int[] { 1 }, TagProtocol.GEN2, null, sensorCodeRead, true, 100);
            r.ParamSet("/reader/read/plan", readPlan);                  // Load the read plan we've defined into the reader
            TagReadData[] sensorReadResults = r.Read(75);               // Read for 75 ms, then stop
            foreach (TagReadData result in sensorReadResults)           // Loop through all tags found and print the EPC,
            {                                                           // frequency, and Sensor Code for each
                string EPC = ByteFormat.ToHex(result.Epc, "", "");
                string frequency = result.Frequency.ToString();
                string sensorCode = ByteFormat.ToHex(result.Data, "", "");
                Console.WriteLine("EPC: " + EPC + " Frequency (kHz): " + frequency + " Sensor Code:  " + sensorCode);
            }

            // Read the On-Chip RSSI Code ==========================================================================================
            // Define a Select command which applies to a Magnus-S2;
            // We want all tags to respond, regardless of their On-Chip RSSI Code value,
            // so our select mask should be a hex value of 1F
            byte[] mask = { Convert.ToByte("1F", 16) };
            // Select USER memory bank and pointer value bit address of hex A0.
            Gen2.Select select = new Gen2.Select(false, Gen2.Bank.USER, 0xA0, 8, mask);
            // The On-Chip RSSI Code is stored in the RESERVED bank, word location hex D
            TagOp onChipRSSIRead = new Gen2.ReadData(Gen2.Bank.RESERVED, 0xD, 1);
            // Define a read plan which uses our Select command and On-Chip RSSI Code read operation
            readPlan = new SimpleReadPlan(new int[] { 1 }, TagProtocol.GEN2, select, onChipRSSIRead, true, 100);
            r.ParamSet("/reader/read/plan", readPlan);                  // Load the read plan we've defined into the reader
            TagReadData[] onChipRSSIReadResults = r.Read(75);           // Read for 75 ms, then stop
            foreach (TagReadData result in onChipRSSIReadResults)       // Loop through all tags found and print the EPC,
            {                                                           // frequency, and On-Chip RSSI Code for each
                string EPC = ByteFormat.ToHex(result.Epc, "", "");
                string frequency = result.Frequency.ToString();
                string onChipRSSICode = ByteFormat.ToHex(result.Data, "", "");
                Console.WriteLine("EPC: " + EPC + " Frequency (kHz): " + frequency + " On-Chip RSSI: " + onChipRSSICode);
            }
        }
        // Example program output:
        // EPC: 002C0000000000000001234 Frequency (kHz): 923250 Sensor Code:  000A
        // EPC: 002C0000000000000001234 Frequency (kHz): 913250 On-Chip RSSI: 0014
    }
}
```

The sample C# code below illustrates how to read the Sensor Code using an Impinj Speedway reader and the Octane API.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Impinj.OctaneSdk;

namespace OctaneSample
{
    class Program
    {
        static ImpinjReader reader = new ImpinjReader();            // Create a new Reader object

        static void Main(string[] args)
        {
            reader.Connect("172.16.1.17");                          // Connect to the reader
            reader.TagOpComplete += OnTagOpComplete;                // Assign an event handler which runs when a tag is read
            reader.DeleteAllOpSequences();                          // Reset the reader
            Settings settings = reader.QueryDefaultSettings();      // Get the default settings
            settings.Antennas.GetAntenna(1).TxPowerInDbm = 20;      // Set antenna 1 to an output power to 20 dBm
            TagReadOp readSensorCodeOp = new TagReadOp();           // Define a read operation for a Magnus-S2 chip
            readSensorCodeOp.MemoryBank = MemoryBank.Reserved;      // Sensor Code is in the Reserved Bank...
            readSensorCodeOp.WordPointer = 0x5;                     // ... at word address hex 5
            readSensorCodeOp.WordCount = 1;
            settings.Report.OptimizedReadOps.Add(readSensorCodeOp); // Load the read operation to the settings object
            settings.Report.IncludeChannel = true;                  // Request channel frequency information
            settings.Report.Mode = ReportMode.BatchAfterStop;
            reader.ApplySettings(settings);                         // Load the settings into the reader
            reader.Start();                                         // Read for 2 seconds, then stop
            System.Threading.Thread.Sleep(2000);
            reader.Stop();
            reader.Disconnect();
        }

        // This is the event handler which prints the results to the console window when the reader reads a tag.
        static void OnTagOpComplete(ImpinjReader reader, TagOpReport report)
        {
            foreach (TagOpResult result in report)
            {
                if (result is TagReadOpResult)
                {
                    TagReadOpResult readResult = result as TagReadOpResult;
                    string EPC = readResult.Tag.Epc.ToString();
                    string frequency = readResult.Tag.ChannelInMhz.ToString();
                    string sensorCode = readResult.Data.ToString();
                    Console.WriteLine("EPC: " + EPC + " Frequency (MHz): " + frequency + " Sensor Code: " + sensorCode);
                }
            }
        }
    }
}
// Example program output:
// EPC: 002C 0000 0000 0000 0000 1234 Frequency (MHz): 924.25 Sensor Code: 0006
// EPC: 002C 0000 0000 0000 0000 1234 Frequency (MHz): 917.25 Sensor Code: 0007
// EPC: 002C 0000 0000 0000 0000 1234 Frequency (MHz): 909.25 Sensor Code: 0008
```

The sample C# code below illustrates how to calculate a CRC−16 word according to the EPC Generation−2 UHF RFID Specification.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;

namespace CRC_16_Example
{
    class Program
    {
        static string CRC16(string dataHexString)
        {
            int numBytes = dataHexString.Length / 2;
            byte[] dataByteArray = new byte[numBytes];    // Convert hex data string into an array of bytes,
            for (int b = 0; b < numBytes; b++)            // with byte 0 corresponding to the least-significant
            {                                             // two hex digits in the string
                dataByteArray[numBytes - 1 - b] = Convert.ToByte(dataHexString.Substring(2 * b, 2), 16);
            }
            BitArray data = new BitArray(dataByteArray);  // Convert the byte array to a BitArray type
            BitArray CRC = new BitArray(16);              // Create the CRC register
            CRC.SetAll(true);                             // Initialize all bits in the CRC register to 1
            for (int j = data.Length - 1; j >= 0; j--)    // This loop simulates the CRC logic described in
            {                                             // Annex F.2 of the EPC Generation-2 UHF RFID Spec
                bool newBit = CRC[15] ^ data[j];          // Data bits are fed into the register MSB first
                for (int i = 15; i >= 1; i--)
                {
                    if (i == 12 || i == 5)
                    {
                        CRC[i] = CRC[i - 1] ^ newBit;
                    }
                    else
                    {
                        CRC[i] = CRC[i - 1];
                    }
                }
                CRC[0] = newBit;
            }
            CRC.Not();
            byte[] CRCbytes = new byte[2];
            CRC.CopyTo(CRCbytes, 0);                      // Convert the CRC BitArray back to a byte array
            string CRCword = Convert.ToString(CRCbytes[1], 16) + Convert.ToString(CRCbytes[0], 16);
            return CRCword;
        }

        static void Main(string[] args)
        {
            string data = "88A7E1477900";
            string CRC = CRC16(data);
            Console.WriteLine(CRC);
        }
    }
}
// This example demonstrates calculating the CRC-16 word when the calibration words in User memory locations
// 0x9, 0xA, and 0xB are 0x88A7, 0xE147, and 0x7900.  The result is 0x8D9F
```

Magnus is a registered trademark of RFMicron, Inc.
Chameleon is a trademark of RFMicron, Inc.
EPC is a trademark of EPCglobal, Inc.

PUBLICATION ORDERING INFORMATION

LITERATURE FULFILLMENT:
Literature Distribution Center for ON Semiconductor
P.O. Box 5163, Denver, Colorado 80217 USA
Phone: 303−675−2175 or 800−344−3860 Toll Free USA/Canada
Fax: 303−675−2176 or 800−344−3867 Toll Free USA/Canada
Email: orderlit@onsemi.com

N. American Technical Support: 800−282−9855 Toll Free USA/Canada
Europe, Middle East and Africa Technical Support:
Phone: 421 33 790 2910
Japan Customer Focus Center
Phone: 81−3−5817−1050

ON Semiconductor Website: **www.onsemi.com**

Order Literature: http://www.onsemi.com/orderlit

For additional information, please contact your local Sales Representative