Axzon Application Note AN002F40

Reading Magnus®-S Sensors

Overview

Magnus®-S RFID chips 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 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 Tag Model Number

The Sensor, On-Chip RSSI, and Temperature Codes are stored at three different word addresses in the tag memory, with the addresses depending on the Tag Model Number of the Magnus®-S chip 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 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 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 instruct all tags to calculate their On-Chip RSSI Code, and select those with an On-Chip RSSI Code within the specified range for subsequent reading.
2. Send a standard Class-1 Generation-2 UHF Read command to retrieve the specific On-Chip RSSI Code for a particular tag which satisfies the power threshold criterion.

Because of this process, it is possible to filter 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 tags respond regardless of their received power, which will be described later in an example.

The On-Chip RSSI Code is generated once the tag receives a Select command with the parameters as described in Table 3. The value is stored in 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 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 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. If the On-Chip RSSI Code is read from a tag which has not received the Select command as described above, the returned value will be 0.

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 chips 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 chips 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 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 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 3 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 tag. Temperature-enabled Magnus®-S3 chips 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 in decimal degrees C) X 10 + 800
CODE2	$A7_h$	12	Temperature Code at the second calibration temperature
TEMP2	$B3_h$	11	(Second calibration temperature in decimal degrees C) X 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
	Field 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
	Field 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
	Field 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
	Field 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.

$$\text{Calibration Temperature in Degrees Celsius} = \frac{\text{TEMPx} - 800}{10}$$

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

$$\text{Temperature in Degrees Celsius} = \frac{1}{10}\left[\frac{\text{TEMP2} - \text{TEMP1}}{\text{CODE2} - \text{CODE1}}(\text{C} - \text{CODE1}) + \text{TEMP1} - 800\right]$$

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.

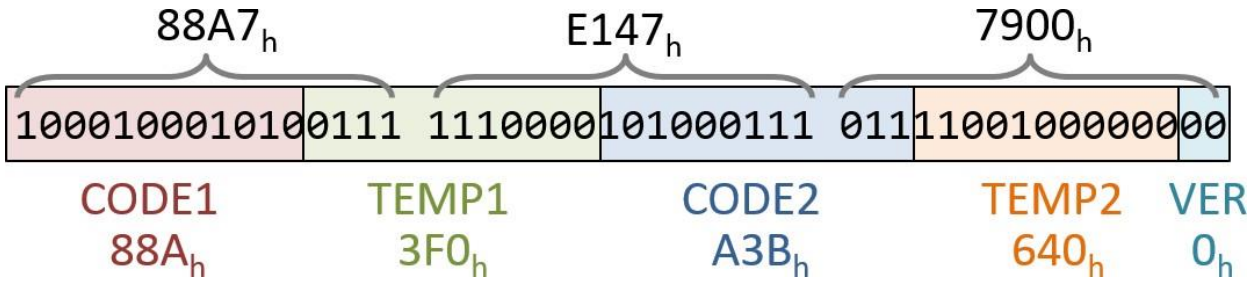

Figure 1 Unpacking calibration words into fields

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.

$$\text{Temperature in Degrees Celsius} = \frac{1}{10}\left[\frac{1600 - 1008}{2619 - 2186}(2315 - 2186) + 1008 - 800\right]$$

$$= 38.44 \text{ degrees C}$$

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:

$$\frac{1008 - 800}{10} = 20.8 \text{ degrees C}$$

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
    {
        public enum SensorType { Magnus2, Magnus3 };
        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
            SensorType sensor = SensorType.Magnus2;                     // Define sensor type as Magnus-2
            // Read the Sensor Code ================================================================================================
            TagOp sensorCodeRead = null;
            // Sensor Code word address depends on Magnus generation
            if (sensor == SensorType.Magnus2)
                sensorCodeRead = new Gen2.ReadData(Gen2.Bank.RESERVED, 0xB, 1);
            else if (sensor == SensorType.Magnus3)
                sensorCodeRead = new Gen2.ReadData(Gen2.Bank.RESERVED, 0xC, 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 frequency = result.Frequency.ToString();
                string sensorCode = ByteFormat.ToHex(result.Data, "", "");
                Console.WriteLine("EPC: " + result.EpcString + " Freq (kHz): " + frequency + " Sensor Code:  " + sensorCode);
            }
            // Read the On-Chip RSSI Code ==========================================================================================
            // 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) };
            Gen2.Select select = null;
            // Select USER memory bank. Pointer bit address depends on Magnus generation
            if (sensor == SensorType.Magnus2)
                select = new Gen2.Select(false, Gen2.Bank.USER, 0xA0, 8, mask);
            else if (sensor == SensorType.Magnus3)
                select = new Gen2.Select(false, Gen2.Bank.USER, 0xD0, 8, mask);
            TagOp onChipRSSIRead = new Gen2.ReadData(Gen2.Bank.RESERVED, 0xD, 1);
            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 freq = result.Frequency.ToString();
                string onChipRSSICode = ByteFormat.ToHex(result.Data, "", "");
                Console.WriteLine("EPC: " + result.EpcString + " Freq (kHz): " + freq + " On-Chip RSSI: " + onChipRSSICode);
            }
        }
        // Example program output:
        // EPC: 000000001234 Freq (kHz): 923250 Sensor Code:  000A
        // EPC: 000000001234 Freq (kHz): 913250 On-Chip RSSI: 0014
    }
}
```


The sample C# code below illustrates how to read the Temperature 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. Change COM port number as needed
            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
            r.ParamSet("/reader/gen2/t4", 3000);                    // Set T4 parameter to 3 ms for best temperature accuracy
                                                                    // This setting will place 3 ms of continuous wave after
                                                                    // the Select command
            // Read the Magnus-3 Temperature Code ================================================================================
            // Define a Select command to activate the temperature engine
            Gen2.Select tempSelect = new Gen2.Select(false, Gen2.Bank.USER, 0xE0, 0, new byte[0]);
            // The Temperature Code is stored in the RESERVED bank, word location hex E
            TagOp tempRead = new Gen2.ReadData(Gen2.Bank.RESERVED, 0xE, 1);
            // Define a read plan which uses the select and read definitions above
            SimpleReadPlan readPlan = new SimpleReadPlan(new int[] { 1 }, TagProtocol.GEN2, tempSelect, tempRead, 100);
            r.ParamSet("/reader/read/plan", readPlan);              // Load the read plan we've defined into the reader
            TagReadData[] tempReadResults = r.Read(75);             // Read for 75 ms, then stop
            foreach (TagReadData result in tempReadResults)         // Loop through all tags found and print the EPC,
            {                                                       // frequency, and Temperature Code for each
                string EPC = result.EpcString;
                string frequency = result.Frequency.ToString();
                string tempCodeHex = ByteFormat.ToHex(result.Data, "", "");
                int tempCode = Convert.ToInt32(tempCodeHex, 16);
                if (tempCode > 1000 && tempCode < 3500)             // Codes outside the operating range should be ignored
                    Console.WriteLine("EPC: " + EPC + " Frequency (kHz): " + frequency + " Temperature Code: " + tempCode);
            }
        }
        // Example program output:
        // EPC: 000000001234 Frequency (kHz): 923250 Temperature Code: 2283
        // The raw Temperature Code of 2275 can be converted to a temperature in degrees
        // by using the calibration information stored in User memory
    }
}
```


The sample C# code below illustrates how to read the Sensor and On-Chip RSSI Code using **Nordic ID handheld RFID readers** and the **Nur API**.

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;
using NurApiDotNet;
using NordicId;
namespace SimpleNordic
{
    public enum SensorType { Magnus2, Magnus3 };
    class Program
    {
        static void Main(string[] args)
        {
            NurApi hApi = new NurApi();
            hApi.ConnectIntegratedReader();                 // Connect to internal reader module
            hApi.Region = NurApi.REGIONID_FCC;              // Set frequency band to FCC (North American region)
            hApi.TxLevel = 10;                              // Set power level; 0=max, 19=min
            NurApi.InventoryExParams InventoryParams = new NurApi.InventoryExParams();
            InventoryParams.inventorySelState = 0;          // These definitions set RFID inventory parameters
            InventoryParams.inventoryTarget = 0;
            InventoryParams.Q = 0;
            InventoryParams.rounds = 5;
            InventoryParams.session = NurApi.SESSION_S0;
            NurApi.TagStorage tagStorage = new NurApi.TagStorage();
            SensorType sensor = SensorType.Magnus3;
            // ============ Read Sensor Code ================================================================================
            if (sensor == SensorType.Magnus2)               // Magnus-2 Sensor Code is in Reserved (Password) bank, word 0xB
                hApi.InventoryRead(true, NurApi.NUR_IR_EPCDATA, NurApi.BANK_PASSWD, 0xB, 1);
            if (sensor == SensorType.Magnus3)               // Magnus-3 Sensor Code is in Reserved (Password) bank, word 0xC
                hApi.InventoryRead(true, NurApi.NUR_IR_EPCDATA, NurApi.BANK_PASSWD, 0xC, 1);
            for (int cycle = 1; cycle <= 20; cycle++)
            {
                try
                {
                    Console.WriteLine("Read attempt " + cycle);
                    hApi.InventoryEx(ref InventoryParams, new NurApi.InventoryExFilter[0]);  // Read tags
                    tagStorage = hApi.FetchTags(true);                                       // Get read results from buffer
                    for (int i = 0; i < tagStorage.Count; i++)
                    {                                       // Sensor Code uses both bytes in Magnus-3 tags
                        int sensorCode = (int)tagStorage[i].irData[0] * 256 + (int)tagStorage[i].irData[1];
                        Console.WriteLine("EPC: " + tagStorage[i].GetEpcString().ToUpper());
                        Console.WriteLine("   Frequency: " + (int)tagStorage[i].frequency);
                        Console.WriteLine("   Sensor Code: " + sensorCode);
                    }
                }
                catch { }
                hApi.ClearTagsEx();                         // Clear contents of tag result buffer
            }
```

Code continues on the next page.

Sample code for Nordic ID handheld RFID readers (continued):

```
            // ============ Read On-Chip RSSI Code ==========================================================================
                                                        // Define Select command to activate On-Chip RSSI calculation
            NurApi.InventoryExFilter onChipRSSI_Select = new NurApi.InventoryExFilter();
            onChipRSSI_Select.bank = NurApi.BANK_USER;
            onChipRSSI_Select.maskBitLength = 8;
            onChipRSSI_Select.action = 0;
            onChipRSSI_Select.target = NurApi.SESSION_S0;
            onChipRSSI_Select.maskData = new byte[1] { 0x1F };// A mask of 0x1F will always match the filter criterion.
            if (sensor == SensorType.Magnus2)
                onChipRSSI_Select.address = 0xA0;             // Select pointer bit address for Magnus-2 is 0xA0
            if (sensor == SensorType.Magnus3)
                onChipRSSI_Select.address = 0xD0;             // Select pointer bit address for Magnus-3 is 0xD0
            NurApi.InventoryExFilter[] filters = new NurApi.InventoryExFilter[1];
            filters[0] = onChipRSSI_Select;
            hApi.InventoryRead(true, NurApi.NUR_IR_EPCDATA, NurApi.BANK_PASSWD, 0xD, 1);
            for (int cycle = 1; cycle <= 20; cycle++)
            {
                try
                {
                    Console.WriteLine("Read attempt " + cycle);
                    hApi.InventoryEx(ref InventoryParams, filters);
                    tagStorage = hApi.FetchTags(true);
                    for (int i = 0; i < tagStorage.Count; i++)
                    {
                        Console.WriteLine("EPC: " + tagStorage[i].GetEpcString().ToUpper());
                        Console.WriteLine("   Frequency: " + (int)tagStorage[i].frequency);
                        Console.WriteLine("   On-Chip RSSI: " + (int)tagStorage[i].irData[1]);
                    }
                }
                catch { }
                hApi.ClearTagsEx();                           // Clear contents of tag result buffer
            }
            hApi.Disconnect();
            Console.ReadLine();
        }
    }
}
```

The sample C# code below illustrates how to read the Temperature Code using **Nordic ID handheld RFID readers** and the **Nur API**

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;
using NurApiDotNet;
using NordicId;
namespace SimpleNordic
{
    class Program
    {
        static void Main(string[] args)
        {
            NurApi hApi = new NurApi();
            hApi.ConnectIntegratedReader();                  // Connect to internal reader module
            hApi.Region = NurApi.REGIONID_FCC;               // Set frequency band to FCC (North American region)
            hApi.TxLevel = 5;                                // Set power level; 0=max, 19=min
            // Build a Select command using the CustomExchange method to activate the Magnus-3 temperature engine.
            // This approach makes it possible to insert a pause in between the transmission of the Select commmand
            // and the rest of the inventory round -- increasing the accuracy of the temperature measurement.
            NurApi.CustomExchangeParams TempSelect = new NurApi.CustomExchangeParams();
            TempSelect.bitBuffer = new byte[NurApi.MAX_BITSTR_BITS / 8];
            int txLen = 0;
            txLen = NurApi.BitBufferAddValue(TempSelect.bitBuffer, 0xA, 4, txLen);  // Command, Select (1010)
            txLen = NurApi.BitBufferAddValue(TempSelect.bitBuffer, 0x0, 3, txLen);  // Target, S0 (000)
            txLen = NurApi.BitBufferAddValue(TempSelect.bitBuffer, 0x1, 3, txLen);  // Action, matching assert SL (001)
            txLen = NurApi.BitBufferAddValue(TempSelect.bitBuffer, 0x3, 2, txLen);  // MemBank, User (11)
            txLen = NurApi.BitBufferAddEBV32(TempSelect.bitBuffer, 0xE0, txLen);    // Pointer, 0xE0
            txLen = NurApi.BitBufferAddValue(TempSelect.bitBuffer, 0x0, 8, txLen);  // Length, 0 bits
            txLen = NurApi.BitBufferAddValue(TempSelect.bitBuffer, 0x0, 1, txLen);  // Truncate, disable
            TempSelect.txLen = (ushort)txLen;
            TempSelect.asWrite = 1; TempSelect.txOnly = 1; TempSelect.noTxCRC = 0;
            TempSelect.rxLen = 0; TempSelect.rxTimeout = 20; TempSelect.appendHandle = 0;
            TempSelect.xorRN16 = 0; TempSelect.noRxCRC = 0; TempSelect.rxLenUnknown = 0;
            TempSelect.txCRC5 = 0; TempSelect.rxStripHandle = 0;

            NurApi.InventoryExParams InventoryParams = new NurApi.InventoryExParams();
            InventoryParams.inventorySelState = 0;           // These definitions set RFID inventory parameters
            InventoryParams.inventoryTarget = 0;
            InventoryParams.Q = 0;
            InventoryParams.rounds = 5;
            InventoryParams.session = NurApi.SESSION_S0;
            NurApi.TagStorage tagStorage = new NurApi.TagStorage();
            hApi.InventoryRead(true, NurApi.NUR_IR_EPCDATA, NurApi.BANK_PASSWD, 0xE, 1);     // Give location of temp code
            for (int cycle = 1; cycle <= 20; cycle++)
            {
                try
                {
                    hApi.Inventory(1, 1, 0);                 // Run a pre-inventory to force a frequency hop
                    hApi.ClearTagsEx();
                    hApi.SetExtendedCarrier(true);           // Turn on continuous wave (CW)
                    hApi.CustomExchange(0, false, TempSelect); // Transmit Select command
                    System.Threading.Thread.Sleep(3);        // Pause for 3 ms while sending CW
                    Console.WriteLine("Read attempt " + cycle);
                    hApi.InventoryEx(ref InventoryParams, new NurApi.InventoryExFilter[0]);  // Read tags
                    tagStorage = hApi.FetchTags(true);                                       // Get read results from buffer
                    for (int i = 0; i < tagStorage.Count; i++)
                    {
                        Console.WriteLine("EPC: " + tagStorage[i].GetEpcString().ToUpper());
                        int temperatureCode = (int)tagStorage[i].irData[0] * 256 + (int)tagStorage[i].irData[1];
                        if (temperatureCode > 1000 && temperatureCode < 3500)                // Use plausible values only
                            Console.WriteLine("   Temperature Code: " + temperatureCode);
                    }
                }
                catch { }
                hApi.SetExtendedCarrier(false);
                hApi.ClearTagsEx();                          // Clear contents of tag result buffer
            }
            hApi.Disconnect();
            Console.ReadLine();
        }
    }
}
```

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 of 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 = 0xB;                     // ... at word address hex B
            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).PadLeft(2, '0') +
                             Convert.ToString(CRCbytes[0], 16).PadLeft(2, '0');
            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 0xBD9F
```


Notices

RFMicron, Inc., ("RFMicron") conditionally delivers this document to a single, authorized customer ("Customer"). Neither receipt nor possession hereof confers or transfers any rights in, or grants any license to, the subject matter of any drawings, designs, or technical information contained herein, nor any right to reproduce or disclose any part of the contents hereof, without the prior written consent of RFMicron.

RFMicron reserves the right to make changes, at any time and without notice, to information published in this document, including, without limitation, specifications and product descriptions. This document supersedes and replaces all information delivered prior to the publication hereof. RFMicron makes no representation or warranty, and assumes no liability, with respect to accuracy or use of such information.

Customer is solely responsible for the design and operation of its applications and products using RFMicron products. It is the Customer's sole responsibility to determine whether the RFMicron product is suitable and fit for Customer's applications and products planned, as well as for the planned application and use of Customer's end user(s). RFMicron accepts no liability whatsoever for any assistance provided to Customer at Customer's request with respect to Customer's applications or product designs. Customer is advised to provide appropriate design and operating safeguards to minimize the risks associated with its applications and products.

RFMicron makes no representation or warranty, and assumes no liability, with respect to infringement of patents and/or the rights of third parties, which may result from any assistance provided by RFMicron or from the use of RFMicron products in Customer's applications or products.

RFMicron represents and Customer acknowledges that this product is neither designed nor intended for use in life support appliances, devices, or systems where malfunction can reasonably be expected to result in personal injury.

This product is covered by U.S. patents 7586385 and 8081043; other patents pending. Chameleon® and Magnus® are trademarks of RFMicron.