

Cresta weather sensor protocol

The Cresta weather station sensors use a 433MHz transmitter to send data to the main station. This document will try to explain what portion of the data I have decoded, and give some pseudo code of the receiving and decoding software which can be used to unravel the data. Since Cresta hasn't released any information about their communication protocol, there may be missing or erroneous information in this description of the protocol.

The serial data stream seems to be transmitted using Manchester coding with a clock of approximately 1 KHz. The clock frequency can of course be derived from the received stream (after all, it is Manchester coded) but I will not do so in this document. Manchester code is basically a method of sending a clock signal combined (xor) with a data signal (Look on the net for more info).

So a simple receiver can be made like this.

```
WaitForInitialTransition()
Data=0
For (i=0; i<NumberOfBits; i=i++)
{
    Data = Data >> 1
    ResetTimer()
    Edge = WaitForNextTransition()
    If (Timer <= ¾ * SignalClockPeriod)
    { /* Short pulses must come in pairs */
        ResetTimer()
        Edge = WaitForNextTransition()
        If (Timer > ¾ * SignalClockPeriod) then error!!! Must be
short!!!
    }
    If (Edge = PositiveEdge) then Data = Data | (1 << HighestBit)
}
```

In this example the least significant bit is received first, because this is the way the sensors send their data. Also, a positive edge translates to a "1", a negative edge translates to a "0". The latter depends, of course, on the receiver output! The code above assumes the output is high if no signal is received.

A Cresta sensor sends 8 bits of data followed by a stop bit for each byte in a stream. The stop bit seems always to be "0". So 9 bits of data are sent per byte, the last bit may be ignored or may be used to check the validity of the data (assuming that it is always "0" of course).

The data packages.

All sensors seem to repeat the same package 3 times during each data transfer. The packages are separated by a pause of at least 10mS but no longer than 50mS. The structure of each package looks like this:

[Byte 0](#): Header byte, always 0x75
[Byte 1](#): Device ID
[Byte 2](#): Package Length and battery status
[Byte 3](#): Device type
[Byte 4..n](#): The actual device data
[Byte n+1](#): Checksum
[Byte n+2](#): Second checksum byte

Bytes 1 to n are encrypted and must be decrypted, using the [routine](#) described at the end of this document, before they can be used. It makes no difference whether the first checksum is checked before or after the decryption as long as you are consistent (i.e. decrypt all, or nothing).

Byte 0: This is easy. It simply is 0x75 for all sensors I have encountered.

Byte 1: This byte identifies the sensor. The table below is a list of the devices I have encountered.

Table1		
ID (Byte 1)	Description	Transmitting interval
0x20 - 0x3F	Thermo/hygro-sensor at channel 1	43 Seconds
0x40 - 0x5F	Thermo/hygro-sensor at channel 2	45 Seconds
0x60 - 0x7F	Thermo/hygro-sensor at channel 3	47 Seconds
0x80 – 0x9F	Rain sensor,	183
	UV sensor or	300
	anemometer	33 Seconds
0xA0 - 0xBF	Thermo/hygro-sensor at channel 4	49 Seconds
0xC0 - 0xDF	Thermo/hygro-sensor at channel 5	51 Seconds

- A sensor selects a random value in the range of column 1 when it is reset. It keeps the same ID until it is reset or the batteries are removed.

Byte 2: Bits 5..1 hold the package length. $n=(b \gg 1) \& 0x1F$. Where **b** is byte 2, and **n** is the number of bytes to read from byte 1 to byte n. So after reading byte 2, we still need to read **n-2** bytes before we reach the checksum.

Bits 7 and 6 remained a mystery until Frank did some 'batbit' hunting and examined bits 7 and 6 some more. He discovered that bits 7 and 6 serve as battery indication bits (the ones I could never find). If the battery voltage drops below a voltage of about 2.5V (Frank has measured 2.485V to be the limit) both bits turn to "0". He also noted that it may take quite some time before a sensor signals it's battery low state.

If the battery status is OK, both bits are "1".

For now, bit 0 remains a mystery and I'll assume it to be "0" until Frank gets out his logic analyzer again and proves me wrong....

Byte 3: Bits 7 and 6 reflect the package number in the stream starting at 01. Bits 4..0 contain the device type as shown in this table. There may be more info in this byte which we haven't discovered yet.

Table2

Byte3 & 0x1F	Device
0x0C	Anemometer
0x0D	UV sensor
0x0E	Rain level meter
0x1E	Thermo/hygro-sensor

Byte 4..n: Are device dependent bytes and will be described in the chapter devoted to that device.

Byte n+1: The checksum byte makes sure that the outcome of an exclusive or of bytes 1..n+1 is 0.

Byte n+2: Is a second checksum byte which is encoded somewhat more exotically. Previously I ignored this byte completely because we don't need it to receive data, (1 checksum is quite enough). But as Dirk Boon pointed out, we do need it in order to send data to a weather station. So thanks Dirk, for giving me the extra work ☺ and of course for your contributions to solving the puzzles. At the [end](#) of this document you will find some notes on sending data to a station. In the C code section you will find a method to calculate the [second checksum](#) byte.

The thermo/hygro-sensor.

The device dependent data of the thermo/hygro-sensor consists of 4 bytes, so n would be 7. This also means that byte 2 must be 0xCE or 0x0E depending upon the battery status.

Byte 4: BCD coded last two digits of temperature. The high nibble holds the units, low nibbles holds the tenths.

Byte 5: Sign and first digit of temperature. High nibble holds 0x4 for negative, 0xC for positive. There may be more info in this nibble (like, whether the sensor measures centigrade or Fahrenheit). The low nibble holds the first digit of the temperature.

Byte 6: BCD coded humidity in percents.

Byte 7: Needs further examination, but it seems to hold flags (bits) for the comfort level. Only the lower nibble (bits 0-3) **seems** to have any meaning. The upper nibble always **seems** to be 0xF. I don't fully understand what bits 0 and 1 of byte 7 mean. They **seem** to indicate that the sensor expects an inaccuracy in its measurements due to sudden temperature changes or a reset. In the case of a sudden temperature change, the body of the sensor needs to heat up or cool down to the air temperature. So the reflected temperature is inaccurate. In case of a reset, the sensor has no memory of the previous measurements and can't tell whether its measurement is accurate. (see tables 3 and 4).

Table3

Bit 1	Bit 0	This is what I think bits 0 and 1 mean. NOT quite sure about this.
0	0	Sensor has been reset. Measurement might be inaccurate.
0	1	Dramatic rise or fall in measurements. Measurement might be inaccurate.
1	0	Recovering. Measurement is probably accurate.
1	1	Situation stable. Measurement is accurate.

Table4

Bit 3	Bit 2	This is what I think bits 2 and 3 mean. Quite sure about this.
0	0	Humidity OK. Temperature uncomfortable. More than 24.9°C or less than 20°C
0	1	Wet. More than 69% RH
1	0	Dry. Less than 40% RH
1	1	Temperature and humidity comfortable.

Here's an example of a thermo/hydro-sensor data transmission. In this example I will use all 3 packages, though they are almost identical. In later examples I'll just use one package.

Package nr.	Raw Data	Decrypted Data
1	75,C3,BA,CA,7D,BF,CF,51,EF	75,45,CE,5E,87,C1,51,F3,EF
2	75,C3,BA,8A,7D,BF,CF,51,AF	75,45,CE,9E,87,C1,51,F3,AF
3	75,C3,BA,4A,7D,BF,CF,51,6F	75,45,CE,DE,87,C1,51,F3,6F

As you can see the temperature was **+18.7** °C and the humidity was **51**% RH in my fridge. These values were transmitted by a thermo/hygro-sensor at channel **2**. This means that Oopsje had left the fridge door open for too long, and my beer was warm! The temperature was *uncomfortable*, in this case that's right, 18.7°C is far too warm☹. Measurement conditions were *stable*. To check the first package: (0xC3^0xBA^0xCA^0x7D^0xBF^0xCF^0x51^0xEF)=0x00, so the checksum of package 1 is OK.

Note that ONLY byte 3 is different for each package, 0x40 is added for each package. This is true for all the devices I have measured.

The Anemometer.

The device dependent data of the anemometer sensor consists of 8 bytes, so n would be 11. This means that byte 2 must be 0xD6 or 0x16 depending upon the battery status.

Byte 4 and 5: These reflect the temperature as measured at the sensor. The value is encoded in the same way as [byte 4 and 5](#) of the thermo/hygro-sensor.

Byte 6 and 7: These reflect the wind chill as measured at the sensor. The value is encoded in the same way as [byte 4 and 5](#) of the thermo/hygro-sensor.

Byte 8: BCD coded. The upper nibble holds the second digit of the wind speed in mph, the lower nibble holds the third digit of the wind speed in mph.

Byte 9: BCD coded. The upper nibble holds the third digit of the wind gust speed in mph, the lower nibble holds the first digit of the wind speed in mph.

Byte 10: BCD coded. The upper nibble holds the first digit of the wind gust speed in mph, the lower digit holds the second digit of the wind gust speed in mph.

Byte 11: High nibble holds encoded direction segment. With encoded I mean that this nibble can be translated to the segment number. Multiplied with 22.5, the segment number gives the direction, in degrees, where the wind comes from. The low nibble holds approach bits in bit 2 and 3 (see table 5). Bits 0 and 1 always seem to be "0".

Table5

Bit 3	Bit 2	Approaches from
0	0	Hasn't moved from this angle.
0	1	Has arrived at this angle via a clockwise rotation.
1	0	Has arrived at this angle via a counter clockwise rotation.
1	1	Never seen this happen.

Example of wind measurement:

Decrypted data: 75,8F,D6,8C,25,C1,24,C1,34,90,03,A8

The **A** of byte 11 translates to 4 using the "[WindDirSeg](#)" routine at the end of this document. This tells us that the wind comes from the direction $4 \times 22.5 = 90^\circ$, this is east. According to the **8** of the same byte it turned to this angle from the south. The gust speed is **03.9** mph, the wind speed is **03.4** mph. The wind chill is **+12.4**°C, the air temperature is **+12.5**°C. And this data looks like a Christmas tree.

The UV index sensor.

The device dependent data of the UV sensor consists of 5 bytes, so n is 8 and byte 2 will be 0xD0 or 0x10 depending upon the battery status.

Byte 4: BCD coded. This **seems** to represent the second and third digit of the temperature at the sensor.

Byte 5: BCD coded. The low nibble seems to represent the first digit of the temperature at the sensor.
The high nibble is the third digit of the MED/h value.

Byte 6: BCD coded. This represents the first and second digit of the MED/h value.

Byte 7: BCD coded. This represents the second and third digit of the UV index value.

Byte 8: The low nibble of this byte probably represents the first digit of the UV index. The high nibble determines the UV level, as shown in the table 6.

Table6

High nibble	UV index	UV level
0	0.0-2.9	LOW
1	3.0-5.9	MEDIUM
2	6.0-7.9	HIGH
3	8.0-10.9	VERY HIGH
4	Above 10.9	EXTREMELY HIGH

Example for UV sensor:

The decrypted data = 75,8F,D0,CD, 07,22,01,28,00

Now we easily see that the temperature at the sensor is 20.7°C, the UV value = 01.2 MED/h, and the UV index = 02.8, the UV level is considered low.

Please note that there is NO temperature sign bit. I have tested it, the sensor reflects the absolute value of the temperature!!

The Rain sensor.

The device dependent data of the rain sensor consists of 3 bytes, so n is 6 and byte 2 will be 0xCC or 0x0C depending upon the battery status.

Byte 4: Binary coded least significant byte of accumulated rain (0.7 mm per unit).

Byte 5: Binary coded most significant byte of accumulated rain (0.7 mm per unit).

Byte 6: Always seems to be 0x66. (My WXR815 doesn't accept anything else).

Example:

The decrypted data = 75,80,CC,8E,D0,00,66,2C

The accumulate rain = 0x00d0 * 0.7mm = 145.6mm. The 66 has an unknown meaning.

Sending data.

In order to send data you will have to calculate the second checksum byte as described below. Also you will have to take into consideration that the receiving station only scans for the packages during about 2 seconds each predefined interval ([See table 1](#)). So the scanning window starts about 1 second before and ends about 1 second after the listed interval. The 1 second before and after is not really tested, it's just an educated guess. If you want to be sure the station accepts your data, transmit with an interval as close as possible to the one listed in [table1](#). While the first checksum can be calculated using raw- or decrypted values, the second checksum needs a different routine for raw- or decrypted data. The routine below works on the raw data.

Written by Ruud v Gessel, April 2009.

Last update, September 2010. Included Frank's battery bit findings (Thanks Frank) and included some [picdemo](#) sources. Added [avr sources](#) for a demo (atmega8 or atmega88) at dec 2010.

Email: dobbeltbeker@chello.nl

See below for decryption routines.

```

/* C code for decryption, and other stuff BYTES are used because the main use will be
   in microcontrollers
*/

typedef unsigned char BYTE;

/* Decrypt raw received data byte */
BYTE DecryptByte(BYTE b)
{
    return b ^ (b << 1);
}

/* Encrypt data byte to send to station */
BYTE EncryptByte(BYTE b)
{
    BYTE a;
    for(a=0; b;
        b<<=1) a^=b;
    return a;
}

BYTE WindDirSeg(BYTE b)
{
    /* Encrypted using: a=-a&0xf; b=a^(a>>1);
       I don't see any easy reversed formula.
       i.e. I can't solve "a" from the equ b=a^(a>>1)
       in one easy single formula. So I solve it
       bit by bit in this method. Does anyone have
       a better solution?
       NOT like this: b &= 0xf;
       return -(b ^ (b >> 1) ^ (b >> 2) ^ (b >> 3))) & 0xf;
       It works... But is, of course, also a bit by bit solution.
    */
    b ^= (b & 8) >> 1; /* Solve bit 2 */
    b ^= (b & 4) >> 1; /* Solve bit 1 */
    b ^= (b & 2) >> 1; /* Solve bit 0 */
    return -b & 0xf;
}

/* The second checksum. Input is OldChecksum^NewByte */
BYTE SecondCheck(BYTE b)
{
    BYTE c;
    if (b&0x80) b^=0x95;
    c = b^(b>>1);
    if (b&1) c^=0x5f;
    if (c&1) b^=0x5f;
    return b^(c>>1);
}

```

```

/* Example to decrypt and check a package,
   Input: Buffer holds the received raw data.
   Returns ERROR number, Buffer now holds decrypted data
*/
#define NO_ERROR 0
#define ERROR_HEADER 1
#define ERROR_CS1 2
#define ERROR_CS2 3

BYTE DecryptAndCheck(BYTE *Buffer)
{
    BYTE cs1,cs2,count,i;

    if (Buffer[0]!=0x75) return ERROR_HEADER;
    count=(DecryptByte(Buffer[2])>>1) & 0x1f;
    cs1=0;
    cs2=0;
    for(i=1; i<count+2; i++)
    {
        cs1^=Buffer[i];
        cs2 =SecondCheck(Buffer[i]^cs2);
        Buffer[i]=DecryptByte(Buffer[i]);
    }
    if(cs1) return ERROR_CS1;
    if(cs2!=Buffer[count+2]) return ERROR_CS2;
    return NO_ERROR;
}

/* Example to encrypt a package for sending,
   Input: Buffer holds the unencrypted data.
   Returns the number of bytes to send,
   Buffer now holds data ready for sending.
*/
BYTE EncryptAndAddCheck(BYTE *Buffer)
{
    BYTE cs1,cs2,count,i;
    count=(Buffer[2]>>1) & 0x1f;
    cs1=0;
    cs2=0;
    for(i=1; i<count+1; i++)
    {
        Buffer[i]=EncryptByte(Buffer[i]);
        cs1^=Buffer[i];
        cs2 =SecondCheck(Buffer[i]^cs2);
    }
    Buffer[count+1]=cs1;
    Buffer[count+2]=SecondCheck(cs1^cs2);
    return count+3;
}

```



```

/* Simple AVR specific routines to send bytes
   TXDDR is the data direction register for the send bit
   TXPORT is the data output port for the send bit
   TX433 is the bit number of the send bit
   In this example bit 5 of portb is connected to the transmitter.
   The controller is running at 8MHz
*/
#include <avr/io.h>
#define F_CPU 8000000UL
#include <util/delay.h>
#define TXDDR DDRB
#define TXPORT PORTB
#define TX433 5

/* Send one byte and keep the transmitter ready to send the next */
void SendManchesterByte(BYTE b)
{
    BYTE i;
    TXPORT |= (1 << TX433);          /* Is always "1" at byte start */
    for (i=0; i<8; i++)
    {
        _delay_us(500);              /* 500uS delay */
        if (b&1)
            TXPORT |= (1 << TX433);
        Else
            TXPORT &= ~(1 << TX433);
        b=~b;
        if (i&1) b>>=1;
    }
}

/* Send bytes (prepared by "EncryptAndAddCheck") and pause at the end. */
void SendManchesterPack(BYTE *Buffer, BYTE cnt)
{
    BYTE i;
    TXDDR |= (1 << TX433);          /* TX433 bit must be output */
    for (i=0; i<cnt; i++)
        SendManchesterByte(Buffer[i]);
    _delay_us(500);                 /* 500uS delay (may be shorter) */
    TXPORT &= ~(1 << TX433);        /* Drop the transmitter line */
    _delay_ms(30);                  /* 30mS delay */
}

/* Send a package as thermo hygro sensor at channel 2,
   t=28.0C, h=51%, comfort flag = 0xff.
*/
void SendTestPackage(void)
{
    BYTE buffer[10], count, tmp;
    for (tmp=0x1e+0x40; tmp>0x40; tmp+=0x40) /* Sends 3 packages */
    {
        buffer[0]=0x75;             /* Header byte */
        buffer[1]=0x40;             /* Thermo-hygro at channel 2 (see table1)/
        buffer[2]=0xce;             /* Package size byte for th-sensor */
        buffer[3]=tmp;              /* Device type = th-sensor, package number */
        buffer[4]=0x80;             /* 8.0 part of 28.0 */
        buffer[5]=0xc2;             /* 2 for 28.0, c for positive temperature */
        buffer[6]=0x51;             /* Humidity 51% */
        buffer[7]=0xff;             /* Comfort flag */
        count= EncryptAndAddCheck(buffer); /* Encrypt, add checksum bytes */
        SendManchesterPack(buffer, count); /* Send the package */
    }
}

```