## E.2) CRC8 Cycle Redundancy Check

**CRC** stands for [Cyclic Redundancy Check](http://en.wikipedia.org/wiki/Cyclic_redundancy_check). It is an error-detecting code used to determine if a block of data has been corrupted. The idea is given a block of N bits, let’s compute a checksum of a sort to see if the N bits were damaged in some way, for instance by transit over a network. The extra data we transmit with this checksum is the “Redundancy” part of CRC, and the second C just means this is a “Check” to see if the data are corrupted (as opposed to an ECC code, which both detects and corrects errors).

**Simple Parity** is another method for error checking for example the number of 1 ‘s and zero’s are even or odd parity for examp 10101010 to be even parity a 1 would be required to append 10101010\_1 to obtain 4 one bits for even parity if s 0 odd parity used a 0 bit would be added. If two bits are switched or lost the error checking will not detect it. Only single bit errors can be detected .

**CheckSum** is another methof for error detection. The conventional 8-bit checksum is just what it sounds like: a sum of all bytes values in the input, with any overflow (from carry operations) discarded. Unfortunately sometimes a CRC value is termed a CheckSum.

**CRC** , treats the message as a big number, we choose a special number to divide the message by (referred to as the “CRC generation polynomial” divisor in the literature), and the remainder of the division is the CRC. Intuitively, it should be obvious that we can detect more than single bit errors with this scheme. Additionally, I think it is obvious that some divisors are better than others at detecting errors. Most implementation do not use division in the normal sense but use Module 2 arithmetic which eliminates the need for the borrowing operation. Modulus 2 arithmetic is XOR exclusive OR operation. For CRC calculations, no normal subtraction is used, but all calculations are done modulo 2. In that situation you ignore carry bits and in effect the subtraction will be equal to an exclusive or operation. This looks strange, the resulting remainder has a different value, but from an algebraic point of view the functionality is equal. A discussion of this would need university level knowledge of algebraic field theory.

The CRC is a predetermined number of bits to be used for the error detection. 8,16,32 or 64 bits are commonly used. This set of notes will concentrate on **CRC8 Dalla/Maxim** 1 Wire usage errors. The number of bits in the error code is n and with CRC8 Dallas/Maxim n = 8.

Why is the predetermined n+1-bit divisor that's used to calculate a CRC called a generator polynomial? Far too many explanations of CRCs actually try to answer that question. This leads their authors and readers down a long path that involves tons of detail about polynomial arithmetic and the mathematical basis for the usefulness of CRCs.

Suffice it to say here only that the divisor is sometimes called a generator polynomial and that you should never make up the divisor's value on your own. Several mathematically well-understood generator polynomials have been adopted as parts of various international communications standards; you should always use one of those. If you have a background in polynomial arithmetic then you know that certain generator polynomials are better than others for producing strong checksums. The ones that have been adopted internationally are among the best of these.

Binary Numbers can be represented as a Polynomial:

BinaryNumber = B[7:0] = B7X7 +B6X6+B5X5 + B4X4+B3 X3 + B2X2 + B1 X1 +B0X0

= B727 +B626+B525 + B424+B3 23 + B222 + B1 21 +B020

B020 = B0\*1

B1 21 = B1 \*2

B222 = B2\*4

B3 23 = B3\*8

B424 = B4\*16

B525 = B5\*32

B626 = B6\*64

B727  = B7\*128

G(X) = B[8:0] = G8X8 + G7X7 +G6X6+G5X5 +G4X4+G3 X3 + G2X2 +G1 X1 +G0X0

G(2) = x8 + x5 + x4 + x0 = 28 + 25 + 24 + 20 = G7G6G5G4G3G2G1G0 = 100110001

**CRC8Maxim Divisor = %1\_0011\_0001 = 13116 = 30510**

**Polynomial Generator** bits 0-8 ( 9 actual bits B8B7B6B5B4B3B2B1B0 )

CRC8Dallas\Maxim = X8 +X5+X4+X0

X8 +X5+X4+X0 = 1\*X8+0\*X7+0\*X6+1\*X5+1\*X4+0\*X3+0\*X2+0\*X1+1\*X0

= %1\_0011\_0001 this is the CRC8 9 bit divisor(Coefficeints)

Note: Polynomial is a shift left multiplier of base 2 = %10

**Endianness**

The endianness is the order of bytes with which data words are stored. We distinguish the following to types:

Little-endian: The least significant byte is stored at the smallest memory address. In terms of data transmission, the least significant byte is transmitted first.

Big-endian: The most significant byte is stored at the smallest memory address. In terms of data transmission, the most significant byte is transmitted first.

**Note:** Parallax propeller is Little Endian processor LSBytes stored in lowest memory address to MSBytes in increasing memory value.

The same conventions can be used in the ordering of the Polynomials . Typically Big Endian convention is mostly used for CRC calculations but little Endian convention can be used.

Endian Example\_1

X8 + X2 + X1 + 1 B[8:0] = B8B7B6B5B4B3B2B1B0

**Big Endian**

G(X) = B8X8+B7X7 +B6X6+B5X5 +B4X4 +B3X3 +B2X2+B1X1+ B0X0

100000111 -🡪 B[8:0] = 100000111

**Little Endian**

G(X) = B0X8+B1X7 +B2X6+B3X5 +B4X4 +B5X3 +B6X2+B7X1+ B8X0

100000111 -🡪 B[8:0] = 111000001

Endian Example\_2

X8 + X5 + X4 + 1 B[8:0] = B8B7B6B5B4B3B2B1B0

**Big Endian**

G(X) = B8X8+B7X7 +B6X6+B5X5 +B4X4 +B3X3 +B2X2+B1X1+ B0X0

100110001 -🡪 B[8:0] = 100110001

**Little Endian**

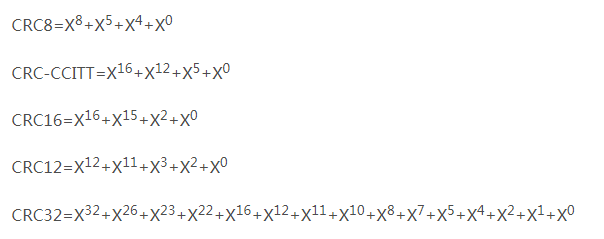
G(X) = B0X8+B1X7 +B2X6+B3X5 +B4X4 +B5X3 +B6X2+B7X1+ B8X0

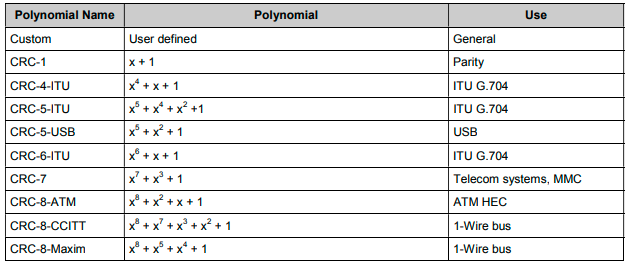
100110001 -🡪 B[8:0] = 100011001

**Note:** That most polynomial specifications either drop the [MSB](https://en.wikipedia.org/wiki/Most_significant_bit) or [LSB](https://en.wikipedia.org/wiki/Least_significant_bit), since they are always 1.

CRC8Dallas/Maxim = $8C = %10001100 or %100011001 adding 1 to LSB

The most commonly used generation polynomials are as follows:





Most Hobbyist usage of CRC values use an 8 bit CRC value and for the remainder of this discussion

CRC-8Maxim will be used. The manufacturer “MAXIM Integrated” now part of Analog devices originally used this for their 1 wire devices which has the CRC8Maxim registers built into their devices.

### E.2.0) CRC Transmission Process

**1) Create DataStream = Data + Checksum(CRC)**

*Divisor*is CRC8Dallas\Maxim = X8 +X5+X4+X0 = 30510 = 13116 = 1001100012

*Data* to be Transferred Let $7778797A = DATA the bytes to be transferred

*Dividend* = DataStream + 8 zeroes (k = number bits in Divisor = n+1 = 9bits)

ADD “n” zero’s (CRC8 is “8+1 = n+1”) so add 8 zero’s to stream to be transferred:

Dividend $7778797A00 = 111\_0111\_0111\_1000\_0111\_1001\_0111\_1010\_0000\_0000

Calculate the CRC (See CRC8 Dalla/Maxim Algorithum)

*CRC* = 10210 = 0x6616 = 1001100012 from CRC calculator n = number bits in CRC = 8

***Data Stream*** = $7778797A66

2**) Transmit Data From Sender to Receiver Device**

Both Transmitter and receiver must be aware of “Generation Polynomial” in this case

CRC8Dallas\Maxim = X8 +X5+X4+X0 = 30510 = 13116 = 1001100012

3**) Receive data and create a receiving end CRC of Data Stream**

When Checking CRC from the receiving end the , the generated CRC is appended to the Data since CRC is a [linear function](https://en.wikipedia.org/wiki/Linear_function) with a property that CRC( xꚚyꚚz)=CRC(x)ꚚCRC(y)ꚚCRC(z).

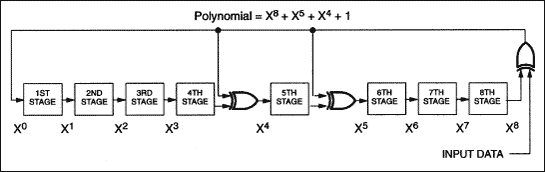
DataStream XOR CRC = 0 in above example 0x66 Ꚛ 0x66 = 0. Doing a CRC on the DataStream if data is good will have a CRC of 0.

### E.2.1) Maxim 1-Wire CRC

The error detection scheme most effective at locating errors in a serial-data stream with a minimal amount of hardware is the CRC. The operation and properties of the CRC function used in Maxim products is presented without going into the mathematical details of proving the statements and descriptions. The mathematical concepts behind the properties of the CRC are described in detail in the references. The CRC can be most easily understood by considering the function as it would actually be built in hardware, usually represented as a shift register arrangement with feedback as shown in **Figure 2**. Alternatively, the CRC is sometimes referred to as a polynomial expression in a dummy variable X, with binary coefficients for each of the terms. The coefficients correspond directly to the feedback paths shown in the shift register implementation. The number of stages in the shift register for the hardware description, or the highest order coefficient present in the polynomial expression, indicate the magnitude of the CRC value that is computed. CRC codes that are commonly used in digital data communications include the CRC-16 and the CRC-CCITT, each of which computes a 16-bit CRC value. The Maxim 1-Wire CRC magnitude is 8 bits, which is used for checking the 64-bit ROM code written into each 1-Wire product. This ROM code consists of an 8-bit family code written into the least significant byte, a unique 48-bit serial number written into the next 6 bytes, and a CRC value that is computed based on the preceding 56 bits of ROM and then written into the most significant byte. The location of the feedback paths represented by the exclusive-OR gates in Figure 2, or the presence of coefficients in the polynomial expression, determine the properties of the CRC and the ability of the algorithm to locate certain types of errors in the data. For the 1-Wire CRC, the types of errors that are detectable are:

1. Any odd number of errors anywhere within the 64-bit number.
2. All double-bit errors anywhere within the 64-bit number.
3. Any cluster of errors that can be contained within an 8-bit "window" (1-8 bits incorrect).
4. Most larger clusters of errors.

The input data is exclusive-OR'ed with the output of the eighth stage of the shift register in Figure 2. The shift register can be considered mathematically as a dividing circuit. The input data is the dividend, and the shift register with feedback acts as a divisor. The resulting quotient is discarded, and the remainder is the CRC value for that particular stream of input data, which resides in the shift register after the last data bit has been shifted in. From the shift register implementation it is obvious that the final result (CRC value) is dependent, in a very complex way, on the past history of the bits presented. Therefore, it would take an extremely rare combination of errors to escape detection by this method.

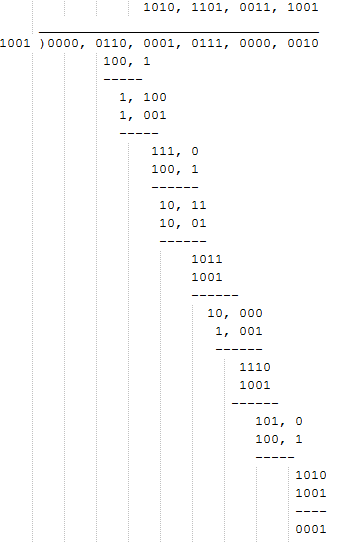
  
***Figure 2. Maxim 1-Wire 8-bit CRC.***

### E.2.1) Modul 2 Binary Division Vs Traditional Division

The basic idea of CRC algorithm is to treat the transmitted data as a very long number of digits.Divide this number by another number.The resulting remainder is appended to the original data as check data.Also take the data from the above example:

6, 23, 4 can be seen as a binary number: 0000011000010111 00000010

If 9 is chosen by dividing, the binary representation is: 1001

Then the division operation can be expressed as:  


As you can see, the last remaining number is 1.If we use this remainder as a checksum, the data transferred is: 6, 23, 4, 1.

The CRC algorithm is a bit similar to this process, but it does not use the usual division in the example above.In the CRC algorithm, binary data streams are used as coefficients of the polynomial, followed by the multiplication and division of the polynomial.Let's give an example.

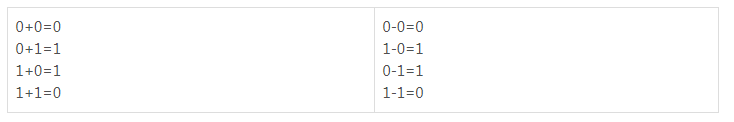
For example, we have two binary numbers: 1101 and 1011.

1101 is associated with the following polynomial: 1x3+1x2+0x1+1x0=x3+x2+x0

1011 is associated with the following polynomial: 1x3+0x2+1x1+1x0=x3+x1+x0

Multiplication of two polynomials: (x3+x2+x0) (x3+x1+x0)=x6+x5+x4+x3+x3+x2+x1+x0

When the result is obtained, the modulo 2 operation is used to merge the same items.That is, multiplication and division use normal polynomial multiplication and division, while addition and subtraction use modulo 2 operations.The so-called modulo 2 operation is to divide the result by 2 and take the remainder.For example, 3 mod 2 = 1.Therefore, the resulting polynomial above is: x6+x5+x4+x3+x2+x1+x0, corresponding to the binary number: 111111

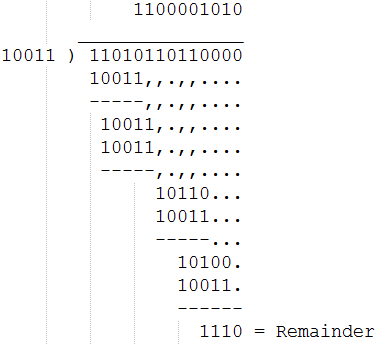
Addition and subtraction with modulo 2 is actually an operation, which is what we usually call XOR:  


As mentioned above, half-day polynomials, in fact, even without introducing the concept of polynomial multiplication and division, can explain the particularity of these operations.Only polynomials are mentioned in almost all the literature explaining the CRC algorithm, so a few basic concepts are simply written here.However, it is very tedious to always use this polynomial representation, and the following instructions will try to use a more concise way of writing.

The division operation is similar to the multiplication concept given above, or the addition and subtraction are replaced by XOR.Here is an example:

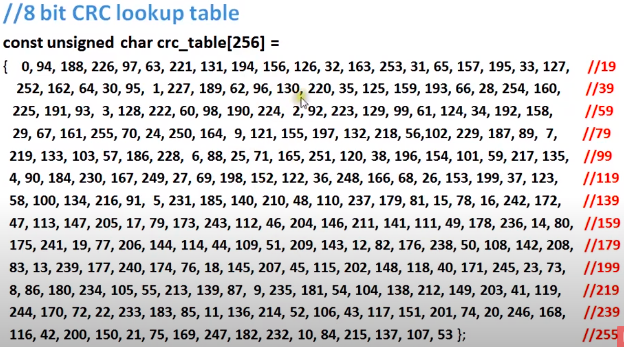
The data to be transferred is: 1101011011

The divisor is set to 10011

Before calculating, four 0:11010110000 are added to the back of the original data, so the reason for adding 0 is explained later.  


From this example, it can be seen that after the addition and subtraction of module 2, the problem of borrowing does not need to be considered, so division becomes simpler.The final remainder is the CRC checkword.In order to perform the CRC operation, that is, this special division operation, a dividend must be specified. In the CRC algorithm, this divider has a special name called "Generate Polynomial".Selection of the resulting polynomial is a very difficult problem. If not, the probability of detecting errors will be much lower.Fortunately, this problem has been studied by experts for a long time. For those of us users, we just need to use the ready-made results.

### E.2.2) CRC8Maxim x^8+x^5+x^4 + 1 Lookup Table



CRC is a [linear function](https://en.wikipedia.org/wiki/Linear_function) with a property that CRC( xꚚyꚚz)=CRC(x)ꚚCRC(y)ꚚCRC(z)

{\displaystyle \operatorname {crc} (x\oplus y\oplus z)=\operatorname {crc} (x)\oplus \operatorname {crc} (y)\oplus \operatorname {crc} (z);}

E.2.3) CRC8 Dallas/Maxim Algorithum

Binary Multiplication

101 5 111001 + 1 = 26 26 mod 5 = 1

X101 x5

101 25

0000

10100

11001 = 25

Binary Division

­\_\_ 101\_\_\_\_\_\_\_\_

101 |11010

101

10

00

100

101

1 Remainder

26 mod 5 = 1

**Modulo 2 Division XOR**

Modulo-2 division is performed similarly to “normal” arithmetic division. The only difference is that we use modulo-2 subtraction (**XOR)**) instead of arithmetic subtraction for calculating the remainders in each step. The quotient is not of interest.

­\_\_111\_\_\_\_\_\_\_\_

101 |11010

101

111

101

100

101

26 mod 5 = 1 1 Remainder

### E.2.4) Types of CRCs

There are different types of CRCs. They are categorized by the degree of the polynomial they use. As the first exponent of a polynomial of degree n is always present by definition (otherwise it would have a lower degree), its binary representation always begins with a 1.

In other words, the first bit of a binary polynomial representation doesn’t carry any information about the polynomial when we agree on a fixed degree.

For that reason, the first bit of a binary polynomial representation is always dropped when computing a CRC in software. So the bit size of the resulting binary is always n for a polynomial of degree n.

It is apparent there is a myriad of CRC implementations and the sending and receiving devices must be using the sam methodology. It is because of this non standardations complexity (obfuscation) rules.

Example:

| Polynomial | Binary Representation | Binary (1st bit dropped) | Bit Size |
| --- | --- | --- | --- |
| x4 + x2 + x + 1 | **10111** | **0111** | 4 |
| x4 + x3 + x2 + 1 | **11101** | **1101** | 4 |
| x8 + x4 + x2 + 1 | **100010101** | **00010101** | 8 |

CRCs types are named by their bit size. Here are the most common ones:

CRC-8

CRC-16

CRC-32

CRC-64

CRC-1 (parity bit) is a special case

Generally, we can refer to a CRC as CRC-n, where n is the number of CRC bits and the number of bits of the polynomial’s binary representation with a dropped first bit. Obviously, different CRCs are possible for the same**n** as multiple polynomials exist for the same degree.

#### E.2.4.1) Error Detection

How do we choose a suitable CRC and a respective polynomial? There are three things we need to consider:

Random Error Detection Accuracy

Burst Error Detection Accuracy

The Redundancy Factor

#### **E.2.4.2) Random Error Detection Accuracy**

Random errors are errors that can occur randomly in any data. For example, a single bit is flipped when transmitting data or a few bits are lost during the transmission.

Depending on the bit size of the CRC we use, we can detect most of these random errors. However, for a CRC-n, 1/2n of these errors cannot be detected. The following table shows the percentage of the possible random errors that remain undetected for each CRC type:

| **CRC Type** | **Undetected Errors** | **% Undetected** |
| --- | --- | --- |
| CRC-8 | 1/28 | 0.39 |
| CRC-16 | 1/216 | 0.0015 |
| CRC-32 | 1/232 | 0.00000002 |
| CRC-64 | 1/264 | 5.4 x 10-20 |

#### E.2.4.3) **Burst Error Detection Accuracy**

Errors in data transmission are oftentimes not random but produced over a consecutive sequence of bits. Such errors are called burst errors. They are the most common errors in data communication.

It’s one of the CRC’s strongest properties to detecting burst errors reliably.

A CRC-n can detect single burst errors with a maximum length of n bits. However, this depends a lot on the polynomial used for computing the CRC. Some polynomials are able to detect multiple bursts of errors in the transmitted data.

| **CRC Type** | **Burst Error Detection** |
| --- | --- |
| CRC-8 | at least a single burst of <= 8 bits |
| CRC-16 | at least a single burst of <= 16 bits |
| CRC-32 | at least a single burst of <= 32 bits |
| CRC-64 | at least a single burst of <= 64 bits |

#### **E.2.4.4) The Redundancy Factor**

Using a CRC for error detection comes at the cost of extra (non-meaningful) data. When we use a CRC-32 (4 bytes), we need to transmit two more bytes of “unnecessary” data as compared to a CRC-16. CRCs with a lower bit size are obviously cheaper with respect to storage space.

Based on these three factors, we can decide which CRC type to choose for our application. However, the polynomial you choose for your CRC also affects the efficiency and quality of your error detection. But that’s a topic for itself and we won’t cover it in this article. Fortunately, there are a couple of standard polynomials used for a particular CRC type and in most cases it makes sense to just use one of these.

### E.2.5) CRC8Dallas\Maxim Algorithum (Rayman thanks for info)

1) Divisor is 10011001 n = 9 bit divisor (CRC bits = n-1)

2) Data bits are to be revesed ordered and add n-1 zeroes (8)

3) Perform Modul 2 division

4) Reverse order of remainder is th CRC8Dallas\Maxim

**Example 1**

**Data** = $C2 = %11000010 **ReverseC2 = >** %01000011 = $43 **Divisor** = 100110001

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

100110001 |01000011**00**000000

100110001

111101**000**

100110001

11011001**0**

100110001

10000011**0**

100110001

0110111**0**  => 0111011 reverse order 8 bit

CRCMaxim($C2) =%01110110 = $76 = 118

**Example 2**

**Data** = $BC = %10111100 **ReverseC2 = >** 00111101 **= $3E** **Divisor** = 100110001

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

100110001 |00111101**000**00000

100110001

11011001**0**

100110001

10000011**0**

100110001

110111**000**

100110001

10001001 => 10010001 reverse order 8 bit

CRCMaxim($BC) =%10010001 = $91 = 118

### E.2.5) Websites for CRC:

<https://rndtool.info/CRC-step-by-step-calculator/> dividend/divisor steps

[https://crccalc.com](https://crccalc.com/)  CRC Calculator different values

<https://www.youtube.com/watch?v=izG7qT0EpBw> provides overview of how CRC derived

<https://www.youtube.com/watch?v=6qAZAX6ymXY> CRC8Maxim and look up table

<https://quickbirdstudios.com/blog/validate-data-with-crc/> CRC8 polynomial generation dropping bit